RN

MOLE™

MICROCONTROLLER
DEVELOPMENT SUPPORT

HPC™ ASSEMBLER/LINKER/LIBRARIAN USER'S MANUAL

Customer Order Number 424410836-001
NSC Publication Number 424410836-001A
August 1987

MOLE™

HPC™ Assembler/Linker/Librarian User's Manual (Preliminary)

REVISION RECORD

REVISION	RELEASE DATE	SUMMARY OF CHANGES
A	08/87	Pilot Release. MOLE™ HPC™ Assembler/Linker/Librarian User's Manual (Preliminary) NSC Publication Number 424410836-001A.

PREFACE

This manual provides information on system programs which support the development of HPC™ microcontroller applications on National Semiconductor's Microcontroller On-Line Emulator (MOLE™) development systems.

Described are:

- ASMHPC
- LNHPC
- LIBHPC

ASMHPC is a cross-assembler for the NSC HPC microcontrollers. This manual describes the format, features and directives. See the *HPC Programmers Manual* for a description of the instructions. *ASMHPC* will accept source output from the HPC C compiler, CCHPC. See the *HPC C Compiler User's Manual*, NSC Publication Number 424410883-001.

LNHPC is a cross-linker which links object files created by *ASMHPC*, to create an absolute object file.

LIBHPC is a cross-librarian which reads object modules produced by *ASMHPC* and combines them into one file called a library.

This manual assumes the user is already familiar with the host operating system. For example, the user needs to know how files are named and used under the operating system. Also the user needs to be able to use an editor to produce symbolic files.

Other applicable MOLE system manuals are:

- *MOLE Brain Board User's Manual*, Customer Order Number 420408188-001.
- *MOLE HPC Personality Board User's Manual*, Customer Order Number 420410447-001.

The information contained in this manual is for reference only and is subject to change without notice.

HPC and MOLE are trademarks of National Semiconductor Corporation.
UNIX is a registered trademark of AT&T.
VMS is a trademark of Digital Equipment Corporation.
MS-DOS is a trademark of MICROSOFT Corporation.

CONTENTS

TABLES

Chapter 1

INTRODUCTION

1.1 OVERVIEW

This manual provides information on system programs which support the development of microprocessor applications on MOLE™ development systems. This manual does not describe all necessary software; for example, there is no information on a text file editor, but such a program is necessary to develop input files for the MOLE cross-assembler, *ASMHPC.*

The HPC cross-assembler, cross-linker and cross-librarian invocation and operation will vary depending on the operating system. In the following chapters, the invocation and operation is described for an MS-DOS™ operating system. See Appendix B for invocation and operation on a UNIX® operating system and Appendix C for invocation and operation on a VMS™ operating system.

Chapter 1 provides an overview of the system programs, *ASMHPC*, *LNHPC* and *LIBHPC*. It also describes the documentation conventions used in this manual.

Chapter 2 describes the inputs, format, directives, controls and outputs for the *ASMHPC* cross-assembler.

Chapter 3 describes the MOLE HPC cross-linker, *LNHPC*.

Chapter 4 discusses the MOLE HPC cross-librarian, *LIBHPC*.

Appendix A lists the ASCII character set in hexadecimal representation.

Appendix B contains invocation and operation for the cross-assembler, linker and librarian on a UNIX operating system.

Appendix C contains invocation and operation for the cross-assembler, linker and librarian on a VMS operating system.

1.2 MOLE HPC CROSS-ASSEMBLER (ASMHPC)

The MOLE HPC cross-assembler (*ASMHPC*) is a cross-assembler for the NSC HPC family of microcontrollers. *ASMHPC* translates symbolic input files into object modules and generates an output listing of the source statements, machine code, memory locations, error messages, and other information useful in debugging and verifying programs.

This manual describes the format, features and directives of the MOLE HPC cross-assembler. See the *HPC Programmer's Manual* for a description of the instructions.

ASMHPC will accept source generated by the HPC C compiler, *CCHPC*. See the *MOLE HPC C Compiler User's Manual*, for more information on *CCHPC*.

1.3 MOLE HPC CROSS-LINKER (LNHPC)

The MOLE HPC cross-linker (*LNHPC*) links object files generated by *ASMHPC*. The result is an absolute load module in various formats, such as, Intel-hex, National "lm" format or COFF (Common Object File Format).

1.4 MOLE HPC CROSS-LIBRARIAN (LIBHPC)

The MOLE HPC cross-librarian (*LIBHPC*) reads object modules produced by *ASMHPC* and combines them into one file called a library. The linker can then search a library for any undefined external symbols.

1.5 DOCUMENTATION CONVENTIONS

The following documentation conventions are used in text, syntax descriptions, and examples in describing commands and parameters.

1.5.1 General Conventions

Nonprinting characters are indicated by enclosing a name for the character in angle brackets < >. For example, <CR> indicates the RETURN key, <ctrl/B> indicates the character input by simultaneously pressing the control key and the **B** key.

1.5.2 Conventions in Syntax Descriptions

Except where otherwise indicated, any combination of upper- and lower-case letters may actually be used when entering commands.

Italics are used for items supplied by the user. The italicized word is a generic term for the actual operand that the user enters.

Spaces or blanks, when present, are significant; they must be entered as shown. Multiple blanks or horizontal tabs may be used in place of a single blank.

{ }	Large braces enclose two or more items of which one, and only one, must be used. The items are separated from each other by a logical OR sign "|."
[ ]	Large brackets enclose optional item(s).
|	Logical OR sign separates items of which one, and only one, may be used.
...	Three consecutive periods indicate optional repetition of the preceding item(s). If a group of items can be repeated, the group is enclosed in large parentheses "( )."
,,,	Three consecutive commas indicate optional repetition of the preceding item. Items must be separated by commas. If a group of items can be repeated, the group is enclosed in large parentheses "( )."

( ) Large parentheses enclose items which need to be grouped together for optional repetition. If three consecutive commas or periods follow an item, only that item may be repeated. The parentheses indicate that the group may be repeated.

␣ Indicates a space. ␣ is only used to indicate a specific number of required spaces.

All other characters or symbols appearing in the syntax must be entered as shown. Brackets, parentheses, or braces which must be entered, are smaller than the symbols used to describe the syntax. (Compare user-entered [], with [ ] which show optional items.)

1.5.3 Example Conventions

In interactive examples where both user input and system responses are shown, the machine output is in regular type. User-entered input is in boldface type. Output from the machine which may vary (*e.g.*, the date) is indicated with italic type.

Additionally, it is a standard of all MOLE documentation to indicate mnemonics in upper-case letters.

Chapter 2

CROSS-ASSEMBLER (ASMHPC)

2.1 INTRODUCTION

This chapter describes the inputs, format, directives and outputs for the *ASMHPC* cross-assembler. See the *HPC Programmers Manual* for a list of instruction mnemonics and functions.

ASMHPC translates source files into object modules which contain instructions in binary machine language. These object modules are linked with *LNHPC* to generate an absolute object; the absolute object module can be loaded into the MOLE Brain/Personality board shared-memory for program debugging and emulation. The *ASMHPC* object modules may also be combined into a library by *LIBHPC*. *ASMHPC* will accept source generated by *CCHPC*, the HPC C compiler.

As an aid to the programmer, the assembler optionally generates an output listing of the source statements, machine code, memory locations, error messages, and other information useful in debugging and verifying programs.

The assembler has a number of directives which are used to control the operation of the assembler. For example, the .TITLE directive controls the identifying title the assembler prints on the pages of the output listings, while the .BYTE directive instructs the assembler to reserve memory bytes.

Section 2.2 describes program invocation.

Section 2.3 describes the elements used in assembler statements:

- Character set
- Location counter
- Rules for symbol and label formation
- Rules for expression evaluation
- Fields used in assembler statements

Section 2.4 describes assembly process.

Section 2.5 describes the assignment statement.

Section 2.6 describes the macros.

Section 2.7 discusses assembler error and warning messages.

Section 2.8 describes the assembler listing.

Section 2.9 describes the assembler directives.

Section 2.10 describes the assembler controls.

2.2 INVOCATION AND OPERATION

This section discusses the invocation of *ASMHPC*, assembler options, search order for include and help files and the default filenames and extensions. See the release letter for installation instructions.

2.2.1 Invocation

The invocation will vary depending on the operating system. See Appendix B for assembler invocation on a UNIX operating system and Appendix C for assembler invocation on a VMS operating system. For the MS-DOS operating system, the invocation line is as follows:

ASMHPC (gives help)

ASMHPC [*asmfile* [,*asmfile* [,...]]] [@*cmdfile*] [@*cmdfile*...] [*options*]

where:

asmfile is the name of a program to assemble. Multiple files may be assembled by joining them with a "," for MS-DOS systems. Multiple source files is designed to allow the user to include standard definition files without having to specify them in each program.

options is a list of assembler options. No options need be specified in which case the assembler defaults will be used. Section 2.2.2 "Assembler Options" describes the option syntax.

cmdfile is the name of a file that contains additional invocation line source filenames and/or options. This file has a default extension of .CMD for MS-DOS systems. For example, if the file *a.cmd* contains:

```
test /a
/o
```

and file *b.cmd* contains

```
/l=file
```

then the command

```
ASMHPC @a /e @b
```

is equivalent to

```
ASMHPC test /a /o /e /l=file
```

Any errors detected on the invocation line causes the assembler to stop execution and display the part in error with an appropriate message.

Some sample invocation lines for MS-DOS systems are:

```
ASMHPC  TEST  /L=CON /NOOBJ/NOTABLE
ASMHPC  JACK.SRC  /CROSS /L
ASMHPC  \DEMO\SAMPLE /D COUNT=6 /PW=120
ASMHPC  \DEMO\SAMPLE /L=\DEMO\
ASMHPC  MATH.DEF,MATH
```

The first command line assembles the file *test.asm* and outputs a listing to the console. No object module or symbol table is produced.

The second command line assembles the file *jack.src* in the current directory and outputs a listing to the default file *jack.lis*. In addition, the object module is written to the file *jack.obj*.

The next example assembles the file *sample.asm* which resides in the directory *demo*. It produces *sample.obj*, which will be in the current directory. The symbol COUNT is defined and given the value 6. Also, the page width is set to 120 characters.

The fourth example is just like the previous one except that the output listing will be written to the same directory as the source file. In this case, the filename is the default which is *sample.lis*.

The last example shows two files being assembled as one assembly file. The program first assembles *math.def*. When the end of file is reached, it then continues the assembly with *math.asm*.

2.2.2 Assembler Options

An assembler invocation line option is an assembler control that is specified in a manner consistent with the operating system. See Appendix B for invocation information for a UNIX system and Appendix C for a VMS system.

The invocation line options for an MS-DOS operating system start with a /, which may be preceded and followed by white space. Assembler options are not case sensitive and may be abbreviated to the minimum number of characters as specified in the control descriptions. For example:

```
/CROSSREF / CNDL
```

See Section 2.10 for a description of all the assembler controls.

2.2.3 Default Filenames and Extensions

For those options that require a filename, the name may include a directory path. It may also consist of just a directory path. In this case the default filename will be used but written to the directory. The default filename is the name of the last source file specified, with any extension removed.

Default extensions depend on the operating system and how the file is specified on the invocation line. For the MS-DOS operating system a default extension is always placed on a file unless one is explicitly specified.

If an output filename consists of just a directory, it should always be terminated by a "\" in an MS-DOS operating system. If not, it will be treated as a filename. Thus, /L=txt\ would output the listing to file *txt\cat.lis* (assuming *cat.asm* is input file). /L=txt would output the listing to *txt.lis*.

2.2.4 Include File Search Order

When searching for a file specified on the .INCLD directive, directories are searched in the following order:

1. current directory
2. directories specified by the /I option
3. default directories
    a. directory specified by environment variable ASMHPC, if it exists
    b. directory specified by environment variable HPC, if it exists
    c. directory \HPC

If the /X option is specified, only directories in category 2 above will be used. Any filename which contains an explicit directory will only be checked for in that directory. No other directories will be searched.

2.2.5 Help File Search Order

When searching for the help file *asmhpc.hlp*, directories are searched in the following order:

1. current directory
2. Default directories (as noted in Section 2.2.4)

2.3 ASSEMBLY LANGUAGE ELEMENTS

This section discusses the format used to write assembler statements.

Specifically:

- Character set
- Location counter
- Symbol and labels
- Expressions
- The four fields of assembly language statements:
    - label field
    - operation field
    - operand field
    - comment field

The statement fields appear in the following order:

label field   operation field   operand field   comment field

Since the assembler accepts free-form statements, the user may disregard specific field boundaries, provided the appropriate delimiters for each field are used (see individual field descriptions). However, for clarity and readability, the use of field boundaries is highly recommended.

2.3.1 Character Set

Each statement is written using the following characters:

Letters — A through Z (a through z)
Numbers — 0 through 9
Special Characters — ! $ % ' ( ) * + , - . / ; : < = > & # b

NOTE: Upper and lower case are distinct, and **b** indicates a blank.

2.3.2 Location Counter

There is a separate location counter for each program section, and the counter is relative to the start of that section. The assembler uses the location counter in determining where in the current program section the current statement goes. For example, if the location counter has the value X'24 (*i.e.,* 24 hex) and the assembler encounters a 1-byte instruction, the assembler will assign the instruction machine code to section address X'24 and will increment the location counter by one since the statement required one byte of memory. If the program section is relocatable, the linker does the final job of assigning an absolute address to the instruction.

The location counter symbol is a single dot (.). If the location counter symbol is used on the right side of an assignment, the left symbol is assigned the current value of the location counter. If the location counter symbol is on the left side of an assignment, the value of the location counter is changed to the value of the right side of the assignment statement.

2.3.3 Symbol and Label Construction

The rules for symbol construction are:

1. The first character of a symbol must be either a letter, a question mark (?), an underline (_), a dollar sign ($) or a period (.).

2. All other characters in the symbol may be any alphanumeric character, dollar sign ($), question mark (?) or underline (_).

Examples:	LOOPloop$_?	legal symbols
	?1	
	_1	
	$1	

3. The first 64 characters are used by the assembler.
4. Symbols which start with a dollar sign are local symbols and are only defined in a local region. (See Section 2.9.25.)
5. Symbols are case sensitive.

Symbols and labels are used to provide a convenient name for values and addresses. The rules for constructing symbol names and the rules for constructing label names are the same; only use distinguishes a symbol from a label. Section 2.3.6 describes how values are assigned to labels. Section 2.5 and Section 2.9.37 describe how values are assigned to symbols.

2.3.4 Operand Expression Evaluation

The expression evaluator in the assembler evaluates an expression in the operand field of a source program. The expressions are composed of combinations of terms and operators. In brief:

- Terms
  - — numbers in decimal, hexadecimal, octal or binary
  - — string constants
  - — labels and symbols
  - — location counter symbol

Each term is composed of four attributes (its value, relocation type, memory type and size). The relocation type is either absolute or relocatable. An absolute term is one in which the value is completely known during assembly. A relocatable term is defined as a label within a relocatable section (see Section 2.9.36), a symbol equated to another relocatable expression or a symbol declared with the .EXTRN directive. The value of a relocatable term is the offset of the label from the start of the section or the external value. Both of these values must be determined by the linker. The memory type of a term indicates whether the term represents a BASE, RAM8, RAM16, ROM8 or ROM16 address. This is also specified by use of the .SECT or .EXTRN directives. In addition, the memory type of a term may be null in the case of an absolute term. The size attribute of a term is null, byte or word. A term has the byte attribute if it is the label on a .DB, .DSB or .FB directive, or it is specified as byte on a .EXTRN or .SET directive or = (assignment). A term has the word attribute if it is the label on a .DW, .DSW, .DL, .DF or .FW directive, or it is specified as word on the .EXTRN or .SET directive or = (assignment).

An expression has the same attributes as a term. They are taken from the terms that comprise it, and only certain combinations of terms are valid in an expression. The relocation type of an expression is derived from its terms as follows (aterm is absolute, rterm is relocatable, op represents any operator, cop represents any conditional operator, *e.g.*, >=):

rterm = rterm + aterm
rterm = rterm - aterm
rterm = aterm + rterm
aterm = aterm op aterm
aterm = rterm - rterm
aterm = rterm cop rterm

In the last two cases, the terms must be relocatable within the same section. Any other expression is considered to have a relocation type of complex and must be resolved by the linker.

The memory or size type of an expression is derived from its terms in a manner similar to the relocation type. If type is an expression with memory or size type and number is an absolute (non-label) value, then the following rules apply:

type = type + number
type = type - number
type = number + type

Any other combinations of memory or size types are considered null. Also, a complex expression is considered to have a null size type.

The notation for the various types of terms is detailed later in this section.

- Operators
  - arithmetic operators
  - logical operators
  - relational operators
  - upper and lower byte extraction operators
  - untype operator

The *arithmetic* operators are the usual +, -, *, /, MOD, SHL, SHR, ROL, and ROR. The *logical* operators are NOT, AND, OR and XOR. The available *relational* operators are EQ, NE, GT, LT, GE and LE. & is the untype operator. The upper and lower byte *extraction* operators are HIGH and LOW. Table 2-1 lists the operators, function and whether the operator is unary or binary. Some operators have optional syntax for compatibility with older assemblers. Table 2-2 lists the precedence order for the evaluation of the operators; a higher precedence operator is evaluated before a lower precedence operator.

Parentheses are permitted in expressions. Parentheses in expressions override the normal order of evaluation, with the expression(s) within the parentheses being evaluated before the outer expressions.

TABLE 2-1. ARITHMETIC, LOGICAL, AND RELATIONAL OPERATORS

OPERATOR	OPTIONAL	FUNCTION	TYPE
+		Addition	Unary or Binary
-		Subtraction	Unary or Binary
*		Multiplication	Binary
/		Division	Binary
MOD		Modulo	Binary
SHL		Shift Left	Binary
SHR		Shift Right	Binary
ROL		Rotate Left	Binary
ROR		Rotate Right	Binary
NOT	%	Logical NOT	Unary
AND	&	Logical AND	Binary
OR	!	Logical OR	Binary
XOR		Logical XOR	Binary
LT	<	"Less Than"	Binary
EQ	=	"Equal To"	Binary
GT	>	"Greater Than"	Binary
LE	<=	"Less Than or Equal To"	Binary
GE	>=	"Greater Than or Equal To"	Binary
NE	<>	"Not Equal To"	Binary
&		Untype	Unary
LOW		Lower 8 bits	Unary
HIGH		High 8 bits	Unary

TABLE 2-2. OPERATOR PRECEDENCE VALUE

OPERATOR	PRECEDENCE VALUE
)	0 (lowest)
NOT, !	1
AND, &	2
OR, %	3
XOR	3
LT, <	4
GT, >	4
EQ, =	4
NE, <>	4
LE, <=	4
GE, >=	4
+	5
-	5
/	6
*	6
MOD	6
SHL	6
SHR	6
ROL	6
ROR	6
LOW	7
HIGH	7
(	8
UNARY -	9 (highest)
UNARY +	9
UNARY &	9
B_SECT	9
E_SECT	9

The assembler recognizes nine types of terms. They are listed with their notations as follows:

1. A decimal constant term is a decimal number that optionally begins with “D'” or “d'”. Leading zero is not permitted for decimal, except for simple case of constant 0.

   Examples: 3, 234, -10, D'3.

2. A hexadecimal constant term is a hexadecimal number that starts with “X'” or “x'” or “H'” or “h'” or “0X” or “0x” or 0. An optional “H” or “h” is permitted at the end.

   Examples: X'23A, H'23A, 0x23A, 023A, 023AH.

3. An octal constant term is an octal number that starts with “O'” or “o'” or “Q'” or “q'”.

   Examples: O'27, Q'27.

4. A binary constant term is a binary number that starts with “B'” or “b'”.

   Examples: B'011, B'0111011.

5. A string constant term is a one or two character string enclosed in single quotation marks.

   Examples: 'Z', '$', '23', '', '''', ''''''.

   The null string '' is evaluated as 0. Within a string, single quote marks are indicated by two quote marks. The string '''' is the single quote mark string, and '''''' is the double quote mark string.

6. The following escape codes may be used in a string constant to represent the value shown:

\a	0x7	(bell)
\b	0x8	(backspace)
\f	0xc	(formfeed)
\n	0xA	(linefeed)
\r	0xD	(carriage return)
\t	0x9	(horizontal tab)
\v	0xB	(vertical tab)
\0	0	(null)
\'	0x27	(quote)
\"	0x22	(double quote)
\\	0x5c	(reverse slash)

   Any lower case character may also be specified as uppercase (*e.g.*, \b and \B are the same).

7. A label term is described in Section 2.3.6 under the label field description.

8. A symbol term consists of a single symbol. The symbol has been given a value by either an assignment statement (Section 2.5) or by the .SET directive (Section 2.9.37).

9. The location counter term is a single dot (.). The dot represents the location counter and, if it appears within an expression, it is replaced by the current value of the location counter.

   Example: JP .

10. A lower-half term is represented by LOW *expression.* An upper-half term is represented by HIGH *expression.* When the assembler encounters one of these in an expression, it replaces it with either the lower or the upper eight bits of the value of the expression.

    Examples: HIGH X'172F is replaced by X'17
    LOW X'172F is replaced by X'2F

11. The size type may be removed from a term with the & operator. For example, if we have

    BYT = 7:BYTE

    then &BYT would have the value 7 and its size type would be null.

12. The B_SECT and E_SECT operators are used to obtain the beginning and ending address of a section respectively. (See .SECT directive Section 2.9.36.) Each has a format of

    B_SECT (*section name*)
    E_SECT (*section name*)

    Example: LD A,#E_SECT(ONE) - B_SECT(ONE)

    will load register A with the length of section ONE.

Numbers are represented internally in the assembler in 16-bit 2's complement notation. The range for numbers in this representation is -32768 (X'8000) to +32767 (X'7FFF) for signed numbers, and 0 to 65535 for unsigned numbers.

String constants are represented internally by the appropriate 8-bit ASCII code (most significant bit is zero).

Examples:		
	''	is replaced by 0
	'A'	is replaced by 041
	'AB'	is replaced by 04142

An expression may consist of a single term, as shown in the following:

```
Examples:  5
           X'3C
           'Q'
           SUB
           .
           HIGH(X'3CF)
           LOW(SUB)
```

Alternatively, an expression may consist of two or more terms combined using the operators shown in Table 2-1.

```
Examples:   36 + SUB
            X'3F0-10
            X'7F AND 'Q'
            3*5 OR XYZ
            (NOT SUB)/2
```

NOTE: All expression evaluation is done treating the terms as unsigned numbers, for example —1 is treated as value X'FFFF.

The magnitude of the expression must be compatible with the memory storage available for the expression. For example, if the expression is to be stored in an 8-bit memory word, then the value of the expression must not exceed X'FF.

2.3.5 Addressing

This section shows the syntax of the various instruction addressing modes, giving examples that use a byte expression or word expression. A byte expression is an expression followed by a .B or an expression that has the size attribute of byte. A word expression is followed by a .W or an expression that has the size attribute of word. A .B or .W force the expression to the byte or word attribute regardless of the type of the underlying expression. The following labels are used in the examples in this section:

```
.SECT example, BASE
WRD1: .DSW 1
BYT1: .DSB 3
NUM = 6
```

NOTE: WRD1 has the word attribute and BYT1 the byte attribute. NUM is of type null. WRD1 and BYT1 are in basepage RAM.

Direct Addressing

A direct operand is specified by a byte or word expression.

```
Example:  LD   A,WRD1+2    ; load contents of word at address WRD1+2
          LD   A,BYT1+1    ; load contents of byte at address BYT1+1
          LD   B,BYT1.W    ; load contents of word at address BYT1
                             (note .W overrides expression type of BYTE)
          LD   A,NUM       ; error since expression has no size type
          LD   A,NUM.B     ; now we know its BYTE type
```

Indirect Addressing

An indirect operand is specified by [*word expression*].W for word indirect; [*word expression*].B for byte indirect. The value of the expression must be an 8-bit address.

```
Example:   LD    A,[B].W         ; word indirect
           LD    A,[WRD1].B      ; byte indirect
           LD    A,[BYT1].W      ; error, indirect address can't be byte expression
           LD    A,[NUM].B       ; warning, null address will default to .W
           LD    A,[WRD1]        ; error, no .B or .W specified
           LD    A,[BYT1.W].B    ; this is valid as .W overrides BYTE type
```

Immediate Addressing

An immediate operand is specified with a #.

```
Example:   LD  A,#BYT1+2   ; load A immediate, byte size is ignored.
           ADD A,#1        ; add immediate to A
```

Register Indirect Addressing, Auto Increment/Decrement

Register indirect is specified by [B+].W, [X+].W for word indirect auto increment and [B+] .B,[X+] .B for byte indirect auto increment. Auto decrement uses — instead of +.

```
Example:   LD  A,[X+].W    ; word indirect, auto increment
           LDS A,[B-].W    ; word indirect, auto decrement
           LDS A,[B-].B    ; byte indirect, auto decrement
```

Indexed Addressing

The indexed operand is specified by *expression*[*word expression*].W for word indexing and *expression*[*word expression*].B for byte indexing. The first expression is the index offset. The [*word expression*] is the 8-bit address of a word. The offset expression is always treated as a null type expression but may not contain a .B or .W explicitly.

```
Example:   LD    A,2[WRD1].W             ; word indexed
           LD    A,BYT1[B].B             ; byte indexed, note use of byte type
                                           expression for offset.
           LD    A,NUM+2[BYT1+8.W].B     ; this is valid as .W override BYTE type
           LD    A,NUM[BYT].B            ; error, indirect address can't be byte
           LD    A,1[WRD1]               ; error, no .B or .W
```

Branch Addressing

The operand of one of the branch type instructions (JP, JMP, JMPL, JSR or JSRL) must be a null type expression and the operand normally is defined in a section that has the ROM16 or ROM8 attribute. Assume we have the following program fragment:

```
        .SECT   ONE,RAM16
DAT:
        .SECT   TWO,ROM16
BYT:    .DB     2
LBL:    LD      A,DAT
```

Then:

```
JMP   LBL     ; is valid instruction
JMP   BYT     ; is an error, operand type must be null
JMP   DAT     ; is a warning, operand is null but not ROM type
JMP   #LBL    ; is valid, same as JMP LBL
JMP   #DAT    ; is valid, # override RAM type
JMP   #BYT    ; valid now, byte type removed by #
```

Operand Size

The assembler normally generates the minimal instruction size possible for each instruction. There may be cases in which the user wants the instruction to be the maximum size for debugging purposes. In these cases, the > operator has been provided to force the maximum size for each operand. This operator may only appear at the start of the operand.

```
Example:   LD   A,>#3          ; will be a 16-bit immediate
           LD   A,>10.B        ; 16-bit direct address
           LD   A,>2[10.W].B   ; 16-bit offset
           LD   >10.W,>#1      ; 16-bit direct and 16-bit immediate values
```

2.3.6 Label Field

The label field is optional and has two uses. Most frequently, the field contains a symbol used to identify a statement referenced by other statements. Symbols used this way are labels. Alternatively, the field contains a symbol whose value is set by the assignment operator. For more information on the second use, see Section 2.5.

When the assembler encounters a label, it assigns the current value of the location counter to the label. A colon (:) is used to delimit (terminate) each label in the label field. (When the label is used in the operand field of an instruction, the colon is omitted.)

The rules for label name construction are the same rules as for any symbol. See Section 2.3.3.

A label referencing an instruction need not be on the same line as the instruction. This allows the programmer, when writing source code, to devote a separate line with comments to labels, providing clearer documentation of the program and allowing for easier editing of the source code.

```
Example:          .=0F000H    ; set location counter to 0F000H.
                              ; label SUB is assigned 0F000H
          SUB:
                  NOP
                  .
                  .
                  .
                  .
```

CAUTION

Read the following before using labels on a blank line.

The assembler always processes labels on a line after it processes any following fields. Therefore, when a label appears on the same line as an assignment statement which alters the location counter, the label is assigned the location counter value after the location counter is altered. A label on a preceding line in this case is not the same value.

```
Example:          .=0F001H    ; set location counter to 0F001H
          SUB:                ; label SUB is 0F001H

          SUBW:   .EVEN       ; label SUBW is aligned to even address 0F002H
          SUB3:               ; label SUB3 is assigned 0F002H
          SUB4:   .=0F010     ; label SUB4 is assigned 0F010H
```

2.3.7 Operation Field

The operation field contains an identifier which indicates what type of statement is on the line. The identifier may be an instruction mnemonic, or an assembler directive. The operation field is required except in lines which consist of only a label and/or comment.

In an instruction statement, the operation field contains the mnemonic name of the desired instruction.

```
Example:    label   operation
            SUB:    NOP
```

In a directive statement, the operation field contains a period (.) immediately followed by the name of the desired directive.

Example: .END

See Section 2.8 for the valid directive names.

In an assignment statement, the operation field contains an equal sign (=). See Section 2.5.

One or more blanks terminate the operation field.

2.3.8 Operand Field

The operand field contains entries that identify data to be acted upon by the operation defined in the operation field, *e.g.*, source or target address for the movement of data, or immediate data for storage or adding to another value.

Some statements do not require use of the operand field. For those that do, the operand field usually consists of one or two expressions (the second expression is separated from the first by a comma.) Expressions can be composed of numbers, string constants, labels and symbols combined with arithmetic, logical and relational operators. Section 2.3.4 describes in detail the types of terms used in expressions, the permitted operators and the order of evaluation used for expressions with more than one operator.

2.3.9 Comment Field

Comments are optional descriptive notes which are printed on the assembler output listing for programmer reference and documentation. Comments should be included throughout the source program to explain subroutine linkages, data formats, algorithms used, formats of inputs processed, and so forth. A comment may follow a statement on the same line, or the comment may be entered on one or more separate statement lines. The comment has no effect on the assembled object module file.

The following conventions apply to comments:

- A comment must be preceded by a semicolon (;).
- All ASCII characters, including blanks, may be used in comments.

2.4 ASSEMBLY PROCESS

The assembler performs its functions by reading through the assembly language statements sequentially from top to bottom, generating the machine code and a program listing as it proceeds. Since it reads statements sequentially, a special problem occurs in most assemblers which must be overcome. If the assembler encounters the statement

```
JMP CLEAR
```

before it encounters the label CLEAR, it will be unable to generate machine code for that instruction. This problem is solved by making the assembler perform two "passes" through the assembly language statements.

Pass 1 of the assembler does not generate an object module or a listing. Its purpose is to assign values to labels and symbols. The assembler assigns labels by using an internal counter called the location counter. Each program section has a separate location counter, which indicates where the current instruction/data is in relation to the start of the section. Each time the assembler encounters an instruction, the location counter is incremented by the size of the instruction. As the assembler encounters program labels, the labels are assigned the current value of the location counter. As symbols are encountered, the symbol is saved with its value; symbols must be defined during pass 1.

During pass 2, the assembler generates the object module and/or listing, as specified by the invocation line. It uses the table of label/symbols to generate machine code values for instructions. It also uses the location counter to determine the section address which each instruction should occupy. The linker does the final job of assigning where each program section will go, and generates the absolute object file.

Although two passes solves the problem previously mentioned, it does not solve it in the optimal fashion. If a microprocessor has two forms of a jump instruction, one that has an 8-bit range and one that has a 16-bit range, it would be up to the user to use the appropriate form of the instruction. However, if the program changed, then some short jumps must be made long ones and vice versa. Also in many assemblers, instructions that reference data have only one opcode, and in the case of forward references there is no choice but to use the long form of the instruction. This is because when the assembler first sees the operand it does not know its size.

Unlike most assemblers, this one has the ability to perform more than two passes and hence can optimize any instructions that use a forward reference. The PASS control allows the user to set the number of passes. In this case the assembler may go through the standard pass 1 phase many times to form the final value of each label.

Throughout this manual, two pass mode refers to the assembler performing a standard two passes. Optimize mode refers to the assembler performing three or more passes. Normally, the assembler will be operating in optimize mode.

2.5 ASSIGNMENT STATEMENTS

Syntax: [*label:*] *symbol* = *expression* [; *comments*]

[*label:*] *symbol* = *expression*:**BYTE** [; *comments*]

[*label:*] *symbol* = *expression*:**WORD** [; *comments*]

Description: The assignment statement assigns the value and all other attributes of the *expression* on the right of the equal sign to the *symbol* on the left of the equal sign. The assignment statement does not generate machine code. It simply assigns the *expression* to a *symbol*. When the *symbol* is used in an instruction statement operand field, the assigned value and attributes are used in code generation. :BYTE or :WORD may be optionally used to assign a size to the symbol. If used, it will override the size of the expression. The & operator may be used to force a null expression. :BYTE may also be specified as :B and :WORD as :W.

```
Example:    Y=5               ; assign the value 5 to Y.
            B1=6:BYTE         ; B1 is a byte expression with a value of 6
```

The assignment statement may also refer to the current value of the location counter. The location counter symbol (".") may appear on both sides of the assignment statement equal sign. If it appears on the left, it is assigned the value of the *expression* to the right side of the equal sign. In that case the *expression* on the right must be defined during the first pass so that the pass 1 label assignments may be made.

```
Examples:   .=X'F020          ; set location counter to address X'F020.
                              ; this is same as .ORG X'F020
            LOC = .           ; save current location counter value in "LOC".
```

NOTE: Care must be used not to put two different instructions at the same location, when altering the location counter.

A symbol may be assigned only one value during an assembly with an assignment statement. Attempting to redefine the value of the symbol will result in an error message. The .SET directive, however, allows symbol values to be redefined during an assembly (see Section 2.9.37.)

Some memory reference symbols are predefined, and may not be redefined. These are the registers SP, A, B, K, X.

2.6 MACROS

Macros make the assembly process easier. Duplicative or similar assembly language statements can be inserted into the program source code instead of manually entering them into the program each time they are required. A macro, once defined, will automatically, during assembly time, place repetitive code or similar code with changed parameters into the assembler source code when called by its macro name. The following sections explain the process of defining and calling macros, with and without parameters, and describe how to use assembler directives related to macro generation.

Using macros, a programmer can gradually build a library of basic routines. Variables unique to particular programming applications can be defined in and passed to a particular macro when called by main programs. Such macros can be automatically included in the assembly source code using the .INCLD directive (Section 2.9.22.)

2.6.1 Defining A Macro

The process of defining a macro involves preparing statements which perform the following functions:

- Giving the macro a name
- Declaring any parameters to be used
- Writing the assembler statements it contains
- Establishing its boundaries

Macros must be defined before they are used in a program. Macro definitions within an assembly do not generate code. Code is generated only when macros are called by the main program. Macro definitions are formed as follows:

```
.MACRO mname [,parameters ]
    .
    .
    .
macro body
    .
    .
    .
.ENDM
```

where:

1. .MACRO is the directive mnemonic which initiates the macro definition. It must be terminated by at least one blank.
2. The *mname* is the name of the macro. It is legal to define a macro with the same name as an already existing macro, in which case the latest definition is used. Previous definitions are, however, retained in the macro definition table; if the existing macro is deleted by the .MDEL directive, the previous definition becomes active. If *mname* is the same name as an instruction mnemonic, the macro definition is used in place of the normal instruction assembly.

   The macro name is used by the main program to call the macro. The name must adhere to the rules given for symbols in Section 2.3.3.
3. The *parameters* are the optional list of parameters used in the macro definition. Each parameter must adhere to symbol rules in Section 2.3.3. Parameters are delimited from *mname* and successive parameters by commas.

   The following are examples of legal and illegal .MACRO directives:

Legal	Illegal	Reason Illegal
.MACRO MAC,A,B	.MACRO SUB,*AB	Special character used
.MACRO $ADD,OP1,OP2	.MACRO 1MAC	First character is numeric
4. The macro body is a sequence of assembly language statements and may consist of simple text, text with parameters, and/or macro-time operators.

5. The .ENDM signifies the end of the macro and must be used to terminate macro definitions.

Simple Macros

The simplest form of macro definition is one with no parameters or macro operators. The macro body is simply a sequence of assembly language statements which are substituted for each macro call. These identical macro calls are inefficient if called repetitively within the same assembly program; a repeatedly used series of assembly language statements within a program should be coded as a subroutine. However, simple macros with no variables are useful in compiling a library of basic routines to be used within different programs. They allow the programmer to simply call the macro within the program rather then repeatedly coding all the macro body statements into each program when needed.

Example:
```
                    ; MACRO "IND1" stores one indirectly into
                    ; memory using B register
        .MACRO IND1 ; begin macro definition
        LD    A,#1
        ST    A,[B].B
        .ENDM
```

Macros With Parameters

The previous macro could be made more flexible by adding parameters to the macro definition. Parameters allow the programmer to specify what is being loaded and stored.

Example:
```
                                  ; MACRO "LOAD" loads DEST with SOURCE
        .MACRO LOAD,DEST,SOURCE   ; DEST is destination, SOURCE is source
        LD    A,SOURCE
        ST    A,DEST
        .ENDM
```

2.6.2 Calling A Macro

Once a macro has been defined, it may be called by a program to generate code. A macro is called by placing the macro name in the operation field of the assembly language statement, followed by the actual value of the parameters to be used (if any). The following form is used for a macro call

mname [*parameters*]

where:	*mname*	is the name previously assigned in the macro definition
	parameters	are the optional list of input parameters. When a macro is defined without parameters, the parameter list is omitted from the call.

A call to the simple IND1 macro, previously defined, would be expanded as follows:

```
Source Program                   Assembled Program

  .                                .
IND1               generates     IND1
  .                              LD    A,#1
  .                              ST    A,[B].B
```

NOTE: The macro call (IND1) as well as the expanded macro opcodes and source code will appear on the assembler listing if the appropriate controls are enabled. The macro call statement (IND1) itself will not generate code.

2.6.3 Using Parameters

The power of a macro can be increased by the use of optional parameters. The parameters allow variable values to be declared when the macro is called. For example, the parameter version of IND1 is LOAD, which can be used to load memory using various addressing modes:

```
Source Program       Assembled Program
  .                    .
LOAD [B].B,#1        LOAD [B].B,#1
  .                  LD   A,#1
  .                  ST   A,[B].B
  .                    .
  .                    .
LOAD 1[254].B,[X].B  LOAD 1[254].B,[X].B
  .                  LD   A,[X].B
  .                  ST   A,1[254].B
  .                    .
  .                    .
```

When parameters are included in a macro call, the following rules apply to the parameter list:

1. One comma and zero or more blanks delimit parameters.
2. A semicolon terminates the parameter list and starts the comment field.
3. Single quotes (') may be included as part of a parameter except as the first character of a parameter.

4. A parameter may be enclosed in single quotes ('), in which case the quotes are removed and the string is used as the parameter. This function allows blanks, commas, or semicolon to be included in the parameter. To include a quote in a quoted parameter, include two quotes ('').

5. Missing or null parameters are treated as strings of length zero.

Parameters Referenced by Number

The macro operator @ references the parameter list in the macro call. When used in an expression, it is replaced by the number of parameters in the macro call. For example, the following .IF directive causes the conditional code to be expanded if there are more than ten parameters in the macro call:

```
.IF @>10
```

When used in conjunction with a constant or symbol (not a macro definition parameter), the @ operator references the individual parameters in the parameter list. The following example demonstrates how this function may define and call a macro to establish a program memory table:

```
.MACRO X
.WORD  @1,@2,@3                           ; first, second, third arguments
.WORD  @Q                                 ; Qth argument
.ENDM

Macro call Generated Code
Q=3                   Q=3
 .                    .
X 3,4,5               X 3,4,5
 .                    .WORD 3,4,5         ; first, second, third arguments
 .                    .WORD 5             ; Qth argument
```

This technique eliminates the need for naming each parameter in the macro definition, which is particularly useful when there are long parameter lists. With the @ parameter count operator, it is possible to create macros which have a variable number of parameters.

2.6.4 Concatenation Operator

The "^" macro operator is used for concatenation. The "^" is removed and the strings on each side of the operator are compressed together after parameter substitution. If the right string is a defined symbol (not a macro definition parameter), the decimal value of the symbol is used.

```
Example:                 .MACRO LABEL,X
            R^X:         .WORD X   ; X is macro parameter
            R^Q:         .WORD Q   ; Q is defined symbol
                         .ENDM

             Macro call  Generated code(without comments)
             Q=1         Q=1
              .           .
             LABEL 0     LABEL 0
              .     R0:  .WORD 0
              .     R1:  .WORD Q
```

Another example of the use of this operation is shown in Section 2.6.7.

2.6.5 Local Symbols

When a label is defined within a macro, a duplicate definition results with the second and each subsequent call of the macro. This problem can be avoided by using the .MLOC directive to declare labels local to the macro definition. The .MLOC directive may occur at any point in a macro definition, but it must precede the first occurrence of the symbol(s) it declares local. Any symbol used before the .MLOC will not be recognized as local. Local macro labels (symbols) appear as ZZ*dddd*, where *dddd* is a particular decimal number.

```
Example:    ; BLOCK MOVE
            ; SOURCE is source, DEST is destination, DESTEND is last dest addr

               .MACRO MOVE,SOURCE,DEST,DESTEND
               LD      X,#SOURCE
               LD      BK,#DEST,#DESTEND
               .MLOC   BMV
            BMV:
               LD      A,[X+],B
               XS      A,[B+],B
               JMP     BMV
               .ENDM
```

```
Source Program       Generated Code
  .                    .
MOVE  4000,40,47     MOVE  4000,40,47
  .                  LD    X,#4000
  .                  LD    BK,#40,#47
  .                  ZZ0000:
  .                  LD    A,[X+].B
  .                  XS    A,[B+].B
  .                  JMP   ZZ0000
  .                    .
MOVE  5000,50,57     MOVE  5000,50,57
  .                  LD    X,#5000
  .                  LD    BK,#50,#57
  .                  ZZ0001:
  .                  LD    A,[X+].B
  .                  XS    A,[B+].B
  .                  JMP   ZZ0001
```

2.6.6 Conditional Expansion

The versatility and power of the macro assembler is enhanced by the conditional assembly directives. The conditional assembly directives allow the user to generate different lines of code from the same macro simply by varying the parameter values used in the macro calls. These directives are described in Section 2.9.21.

Example:

```
; if add flag <>0, add X to A; else subtract X from A
.MACRO ADDSUB ADDFLG,X
.IF ADDFLG
ADD A,#X
.ELSE
ADD A,# -X
.ENDIF
.ENDM
```

2.6.7 Macro-Time Looping

The following examples show the use of the .DO, .ENDDO, .EXIT directives. The macro CTAB generates a constant table from 0 to MAX where MAX is a parameter of the macro call. Each word has label "DY":, where Y is the decimal value of the data word:

```
        .MACRO CTAB,MAX
        .SET   Y,0
        .DO    MAX+1
  D^Y:  .WORD  Y
        .SET   Y,Y+1
        .ENDDO
        .ENDM
      Source assembly         Assembled code
        .                     .
        CTAB 2                CTAB 2
          .                   .SET Y,0
          .                   D0: .WORD Y
          .                   .SET Y,Y+1
          .                   D1: .WORD Y
          .                   .SET Y,Y+1
          .                   D2: .WORD Y
          .                   .SET Y,Y+1
```

2.6.8 Nested Macro Calls

Nested macro calls are allowed; that is, a macro definition may contain a call to another macro. When a macro call is encountered during macro expansion, the state of the macro currently being expanded is saved and expansion begins on the nested macro. Upon completing expansion of the nested macro, expansion of the original macro resumes. The allowed number of levels of nesting depends on the sizes of the parameter lists, but at least ten is typical.

A logical extension of a nested macro call is a recursive macro call, that is a macro that calls itself. This is allowed, but care must be taken not to generate an infinite loop.

2.6.9 Nested Macro Definitions

A macro definition can be nested within another macro. Such a macro is not defined until the outer macro is expanded and the nested .MACRO statement is executed. This allows the creation of special-purpose macros based on the outer macro parameters and, when used with the .MDEL directive, allows a macro to be defined only within the range of the macro that uses it.

2.6.10 Macro Comments

Normally all lines within a macro definition are stored with the macro. However, any text following ";" is removed before being stored. A line that starts with ";" is completely removed from the macro definition. These lines will appear on the listing of the macro definition, they will not appear on an expansion.

2.7 ERROR AND WARNING MESSAGES

When an error or warning message occurs, the "^" symbol will generally point at, or just after, where the error occurred.

```
Example:     LD    A,[258].W
                        ^
            ERROR, VALUE OUT OF RANGE
```

2.7.1 Command Line Errors

Assembler errors are divided into command line errors and assembly time errors. For a command line error, the message is displayed after the invocation line. The file in error or the part of the line in error is also shown. A pointer indicates the position of the error. The command line errors are:

Invalid File Name
: the filename is not valid for MS-DOS.

File Conflict
: the filename shown is being used multiple times as an output file or an output filename is the same as the source filename.

File Not Found
: the filename shown cannot be found. Possibly the wrong extension has been assumed or it resides in a different directory.

Disk/Dir full
: no more room exists to create an output file.

No Source File Specified
: the command line must contain at least the source filename.

Device Can't be source
: the source file must be a disk file. It cannot be a device such as the console (CON) or printer.

Invalid Command
: a control specified on the command line is invalid or one of its operand is bad. The part in error is pointed to.

2.7.2 Assembly Time Errors

Each assembly time error is shown with its number, message, and the conditions that cause the error (see Table 2-3). Errors are formatted as follows:

```
#          message
```

where:	#	is the error number.
	message	is the error message that displayed on the output listing.

TABLE 2-3. ASSEMBLER ERRORS

MESSAGE NUMBER	MESSAGE AND CAUSES
1	Invalid or Missing Opcode number is next token after label delimiter after label is not comment or EOL opcode has bad terminator
2	Undefined Opcode opcode token not in opcode tables
3	Label error delimiter is first character on line invalid local symbol
4	Bad Label Terminator bad label terminator
5	Duplicate Label/Symbol duplicate label symbol is already defined .SET symbol already defined as a non .SET symbol external symbol in .PUBLIC external symbol already defined duplicate formal parameter
6	Missing Label =, .SET, missing label no label on .MACRO
7	Undefined Symbol undefined symbol .PUBLIC symbol not defined macro not defined in .MDEL .SPT entry is undefined
8	Argument error invalid .EXTRN type

TABLE 2-3. (Cont)

MESSAGE NUMBER	MESSAGE AND CAUSES
9	Syntax error bad operator illegal op combination
10	Illegal/Invalid Character bad character in line; it is replaced by ^
11	Invalid Numeric number not valid for radix
12	Value Out of Range byte value is out or range or relocatable .DS value too big alignment > section maximum address .SECT directive option value out of range .IPT entry out of range
13	Invalid Register
14	Missing or Bad Symbol .PUBLIC, .EXTRN, not a symbol option in .SECT not a symbol or after = not a symbol formal parameter not a symbol bad operand or .MDEL
15	Missing Operand missing instruction or directive operand
16	Missing or Bad Separator .IFSTR, error in string bad actual parameter macro string operator error missing separator between formal parameters
17	Missing or Bad Delimiter

TABLE 2-3. (Cont)

MESSAGE NUMBER	MESSAGE AND CAUSES
18	Invalid Operand .EXTRN type is invalid bad .SECT option or hit terminator looking for option .MDEL directive operand is bad missing EQ or NE in .IFSTR
19	Multiple Externals two externals in a non-complex expression
20	Two Operands in Sequence an expression has two operands without an intervening operator
21	Missing Keyword
22	Missing String delimiter a string type operand is missing the terminating delimiter
23	Invalid Keyword Usage
24	Nesting error too many conditional levels multiple .ELSE's in conditional block conditional block still open at end of program too many relocatable sections too many macro calls open conditional block on macro exit
25	Questionable Operand Combination the combination of operands is not valid for the instruction
26	Forward Reference
27	Invalid Addressing Mode have [value.B] operand
28	Relocation Usage error an operand that must be absolute is relocatable relocatable operand combination is invalid for operator

TABLE 2-3. (Cont)

MESSAGE NUMBER	MESSAGE AND CAUSES
29	Value not HIGH/LOW - forced LOW non-BASE relocatable value
30	Invalid External Usage .= or .ORG contains external = operand is external but not the only token .END is external .PT is external
31	Branch Out of Range JP, JMP, JSR, or .PT operand is out of range
32	Trailing Characters Extra characters were found at the end of this line; maybe it should be a comment.
33	Phase error The label had a different value during Pass 1. The location counter will be set to the label value. Some instructions must have a forward reference.
34	String in Expression too Large
35	Unrecognized Control
36	"NO" not Valid for Control this control is not allowed to use NO
37	Can't have Primary Control A primary control can only be used on the command line or at beginning of program.
38	Bad File Name The filename in this control is invalid.
39	Can only be on Command Line This control is only valid on the command line.

TABLE 2-3. (Cont)

MESSAGE NUMBER	MESSAGE AND CAUSES
41	Program Counter Overflow/Underflow The location counter is greater than the maximum section size. In an absolute section the location counter cannot be smaller than the size set with the .SECT directive.
42	Source File can't be a Device The file specified in an .INCLD must be a file.
43	File Not found
44	Invalid Local symbol usage The local symbol has the wrong format or a bad numeric.
45	Section error .= or .ORG has a relocatable operand that doesn't match the current program section .PT operand not in same section as directive JP, JMP, JSR address in different section
46	Options don't match previous usage An option used on the .SECT directive doesn't match the options from a previous usage. These options are ignored.
47	Opcode Usage error .ELSE, .ENDIF not in conditional .MLOC, .EXIT, .ENDM used outside of macro
48	Invalid Character in Expression An expression has a character that can't be part of an expression.
49	Attribute Conflict The attributes specified on the .SECT directive conflict. For example both ABS and REL can't be specified for section.
50	User Error user specified via .ERROR directive
51	User Warning user specified via .WARNING directive

TABLE 2-3. (Cont)

MESSAGE NUMBER	MESSAGE AND CAUSES
52	Invalid Processor Usage This instruction is not valid for the processor.
53	Invalid Index Register The index register is invalid for instruction.
54	Expression too Big The complex expression is too large for the assembler to process.
55	Invalid Complex Usage This operand may not be complex.
56	Too Many Parameters Only 125 formal parameters may be defined, there are more actual parameters than formals for this macro.
57	Not Valid in Absolute Mode
58	Instruction or Directive not valid in Segment Object code may not be generated in a section with this segment type.
59	Absolute Value Required An operand of this instruction or directive must be absolute.
60	Ambiguous Control An invocation line or control line option (control) is not unique. Specify additional letters for the option.
200	Invalid Alignment for Section or Directive The directive can't be in a Byte aligned section.
201	.SPT Table is Full attempt to use .SPT more than 16 times
202	Reference must be Even A direct memory word reference or an indirect address must be an even address.

TABLE 2-3. (Cont)

MESSAGE NUMBER	MESSAGE AND CAUSES
203	NO .B or .W Defaults to .W
204	Invalid Alignment for Section or Directive same as 200 but this is a warning
205	Section not Specified A .SECT directive must be used before any object code or storage can be allocated.
206	Divide by 0
207	.B or .W not valid for Operand, Ignored .B or .W are not valid for this operand, .B or .W are ignored
208	Missing Options, Defaults will be used On a .SECT directive at least the section name and ROM16/ROM8/RAM16/RAM8/BASE must be specified.
209	Invalid Floating Point Value A floating point value can only be written in the format described in the .DF directive.
210	Floating Point Over/Underflow The specified floating point value was too large or too small to fit in the IEEE 32-bit floating point format.
211	Byte type not valid for expression. The specified expression may not have an implied byte type or use .B.
212	Word type not valid for expression The specified expression may not have an implied word type or use .W.
213	Expression not ROM type the specified operand defined in non-ROM section

TABLE 2-3. (Cont)

MESSAGE NUMBER	MESSAGE AND CAUSES
214	DEBUGGER DIRECTIVE: Too Many Dimensions Too many dimensions were specified on the .DIM directive.
215	DEBUGGER DIRECTIVE: Invalid Storage Class The value on the .SCL directive is invalid.
216	DEBUGGER DIRECTIVE: Definition not in effect or didn't finish last one. A debugging directive has been specified but a .DEF directive was not specified. Another .DEF directive has been read but a .ENDEF did not finish the last one.
217	Missing .B or .W This operand requires a .B or .W but it is missing.
218	RAM8/ROM8 or Byte aligned Section will not allow word reference Data is RAM8 or ROM8 or byte aligned section may be accessed as word.
219	Byte destination used with 16 bit immediate value A warning which indicates an instruction has a byte destination with a 16 bit source *e.g.*, LD Data.B,#0x1FFF.
220	Word reference to RAM8/ROM8 or Byte Section A word reference is made to a RAM8 or ROM8 section or to a byte aligned section.

2.8 THE ASSEMBLY LISTING

The listing contains program assembly language statements, together with line numbers and page numbers, error messages, and a list of the symbols used in the program.

The listing of assembly language statements which generate machine code includes the hexadecimal address of memory locations used for the statement and the contents of these locations. To the left of the instruction, an "R" indicates a relocatable argument in this instruction, "X" indicates an external argument, "C" indicates a complex argument and "+" indicates macro expansion.

The assembler listing optionally includes an alphabetical listing of all symbols used in the program together with their values, absolute or relocatable type, word or byte or null type, section memory type and public or external.

Optionally a cross reference of all symbol usage by source line number is given; the defining line number is preceded by a "-".

The total number of errors and warnings, if any, is printed with the listing. Errors and warnings associated with assembly language statements are flagged with descriptive messages on the appropriate statement lines.

2.9 DIRECTIVES

Directive statements control the assembly process and may generate data in the object program. The directive name may be preceded by one or more labels, and may be followed by a comment. The directive's name occupies the operation field. Some directives require an operand field *expression*.

Assembler directive statements and their functions are summarized in Table 2-4. All directive statements begin with a period.

The .COMPILED directive is reserved for use by the HPC C compiler; it should not be used.

TABLE 2-4. SUMMARY OF ASSEMBLER DIRECTIVES

DIRECTIVE	FUNCTION	SECTION
.BYTE	8-bit data generation	2.9.1
.CHIP	Specify member of HPC family	2.9.2
.CONTRL	Automatic code alteration control	2.9.3
.DB	8-bit data generation	2.9.1
.DF	Floating data generation	2.9.4
.DL	32-bit data generation	2.9.5
.DO	Macro loop directive	2.9.6
.DOPARM	Macro loop directive	2.9.7
.DSB	Reserve 8-bit data	2.9.8
.DSF	Reserve floating point data	2.9.8
.DSL	Reserve 32-bit data	2.9.8
.DSW	Reserve 16-bit data	2.9.8
.DW	16-bit data generation	2.9.41
.ELSE	Conditional assembly directive	2.9.9
.END	End of source program; reset address	2.9.10
.ENDDO	Macro loop end	2.9.11
.ENDIF	Conditional assembly directive	2.9.12
.ENDM	End macro	2.9.13
.ENDSECT	End program section	2.9.14
.ERROR	User error message	2.9.15
.EVEN	Force even location counter	2.9.16
.EXIT	Macro loop exit	2.9.17
.EXITM	Macro loop termination	2.9.17
.EXTRN	Externally defined symbols	2.9.18
.FB	Fill bytes	2.9.19
.FW	Fill words	2.9.19
.FORM	Output listing top-of-form	2.9.20
.IF	Conditional assembly directive	2.9.21
.IFB	Conditional assembly directive	2.9.21

TABLE 2-4. (Cont)

DIRECTIVE	FUNCTION	SECTION
.IFC	Conditional assembly directive	2.9.21
.IFDEF	Conditional assembly directive	2.9.21
.IFNB	Conditional assembly directive	2.9.21
.IFNDEF	Conditional assembly directive	2.9.21
.IFSTR	Conditional assembly directive	2.9.21
.INCLD	Include disk file source code	2.9.22
.IPT	Interrupt table generation	2.9.23
.LIST	Listing output control	2.9.24
.LOCAL	Establish a new local symbol region	2.9.25
.MACRO	Macro directive	2.9.26
.MDEL	Macro delete directive	2.9.27
.MLOC	Macro local directive	2.9.28
.ODD	Force odd location counter	2.9.29
.OPDEF	Define opcode	2.9.30
.ORG	Set location counter	2.9.31
.OUT	Message to console	2.9.32
.OUT1	Message to console	2.9.32
.OUT2	Message to console	2.9.32
.OUTALL	Message to console	2.9.32
.PT	JID table generation	2.9.33
.PTW	JIDW table generation	2.9.34
.PUBLIC	Public symbols	2.9.35
.SECT	Define program section	2.9.36
.SET	Assign values to symbols	2.9.37
.SPACE	Space n lines on output listing	2.9.38
.SPT	Single byte subroutine table generation	2.9.39
.TITLE	Identification of program	2.9.40
.WARNING	User warning message	2.9.15
.WORD	16-bit data generation	2.9.41

2.9.1 .byte, .db Directives

Syntax: [*label:*] **.BYTE** *expression* [*,expression...*][; *comments*]

[*label:*] **.DB** *expression* [*,expression...*][; *comments*]

Description: The .BYTE and .DB directives generate consecutive 8-bit bytes of data for each given *expression*. If the directive has a *label*, it refers to the address of the first *byte*. The value of each *expression* must be in the range -128 to +127 for signed data or 0 to 255 for unsigned data. The .BYTE and .DB directives are only valid in a ROM type section. Any label will be assigned the byte type.

The hexadecimal value of ASCII characters may be stored in memory using the .BYTE (and .DB) directive and an operand *expression* specifying character strings or their hexadecimal equivalents. (See Appendix A.)

NOTE: A single quote mark in a string is represented by two quote marks. An ASCII character may also be specified using the escape characters described in Section 2.3.4.

Examples:

```
1.      .BYTE   X'FF
2.  T:  .BYTE   MPR-10, X'FF
3.      .BYTE   'DON''T'
4.      .DB     X'44,X'4F,X'4E,X'27,X'54
```

Example 1 stores the hexadecimal number FF in a byte of memory.

Example 2 stores two hexadecimal numbers in consecutive bytes in memory.

Examples 3 and 4 store the ASCII string (DON'T) in consecutive bytes of memory.

2.9.2 .chip Directive

Syntax: [*label:*] **.CHIP** *string* [; *comments*]

Description: The .CHIP directive is for future use, to specify the member of the HPC family. The *string* is up to eight alpha numeric characters, and is put into the load module output file. The .CHIP string in different assembly source modules should not conflict.

Example: .CHIP 16040

2.9.3 .contrl Directive

Syntax: [*label:*] .CONTRL *expression* [*; comments*]

Description: The .CONTRL directive controls automatic code alteration by the assembler on instructions. This directive either decreases the number of bytes of the instruction (to optimize code) or increases the number of bytes of the JP, JMP and JSR instructions to avoid a range error. Note that in two pass mode, the assembler can only alter those instructions which have their operand defined during pass 1, which excludes forward-reference operands. This is because the assembler must know the size of the instruction on pass 1. In optimize mode, the assembler can always choose the optimal size for these instructions.

The .CONTRL directive also allows maximum code size for all instructions to be selected for ease of patching code during debugging. The instructions will all have maximum size operand fields, so any valid operand can be patched in during debugging.

Control of the various options depends upon the three least significant bits of the evaluated expression in the operand field (the expression must be defined during pass 1). Table 2-5 shows the options available, their associated bit weights and assembler default values. Table 2-6 shows the possible code alterations that may occur for the jump instructions.

The .CONTRL directive may be used throughout the program, to enable or disable the alteration of code. Normally, it would be desirable to disable code alteration of the instructions only during a block of code which must remain a fixed size (for example, a critical timing loop.)

This directive takes precedence over any multi-pass optimization. Thus if code reduction is disabled, no optimization will take place regardless of the number of passes.

Example:

```
.CONTRL 0     ; disable all code alteration
     .        ; fixed size code block
     .
     .
.CONTRL 3     ; re-enable code alteration
```

For further code optimization notes, see Section 2.7.

TABLE 2-5. .CONTRL OPTIONS

CONTROL FUNCTION	BIT POSITIONS	BINARY VALUE	3-BIT HEX VALUE	DESCRIPTION
Reduce Code	0	0	0	Suppress reduce code
(Optimize)		1	1	*Enable reduce code
Increase Code	1	0	0	Suppress increase code
(Prevent range errors)		1	2	*Enable increase code
Maximum Code	2	0	0	*Enable bit 0,1 values
(For debug patching)		1	4	Force maximum code
The * indicates default.				

TABLE 2-6. CODE ALTERATION BASED FOR JP, JMP, JMPL, JSR, JSRL

CODED INSTRUCTION	OPERAND IN JP RANGE (1 BYTE OPCODE RESULT)	OPERAND IN JMP RANGE (2 BYTE OPCODE RESULT)	OPERAND IN JMPL RANGE (3 BYTE OPCODE RESULT)
JP	X	+	+
JMP	-	X	+
JMPL	-	-	X
	OPERAND IN .SPT DIRECTIVE	OPERAND IN JSR RANGE	OPERAND IN JSRL RANGE
JSR	-	X	+
JSRL	-	-	X

Bit 2 must = 0
- is possible result if reduce code (bit 0) = 1
+ is possible result if increase code (bit 1) = 1
X is unchanged result, possible whether bit 0, bit 1 = 1 or 0

Example:

```
                                      ; default .CONTRL in effect
                                      ; (code alteration enabled)
       BCKWRD:                        ; label
                RET
                                      ; note that following references to
                                      ; BCKWRD are not forward-references
                JMP     BCKWRD        ; reduced to single byte (JP)
                JP      BCKWRD        ; no change (JP)
                                      ; note that following references to
                                      ; FORWRD are forward references
                                      ; assembler cannot alter these in two
                                      ; pass mode but can in optimize mode
                JMP     FORWRD        ; no change (JMP) in two pass, else (JP)
                JP      FORWRD        ; no change (JP)
       FORWRD:
```

2.9.4 .df Directive

Syntax: [*label:*] **.DF** *value* [*,value...*] [*; comments*]

Description: The .DF directive generates consecutive 32-bit floating point values in standard IEEE format. The 4 bytes of the value are stored starting with the low byte of the mantissa in low memory. Any label will be given the word attribute. This directive may only appear in a ROM type section. The value must be a single floating point number, an expression involving floating point values is not valid.

If the program section is ROM16 and word aligned, .DF is aligned to an even address (along with label), with a zero byte in the skipped byte.

Example:

```
1.          .DF    -11.375
2. FLT1:    .DF    1.0,10.0,1.E-3
```

Example 1 will store the four bytes 0x00, 0x00, 0x36 and 0xC1 in consecutive memory locations.

Example 2 will store the 3 floating point values in consecutive locations.

2.9.5 .dl Directive

Syntax: [*label:*] .DL *value* [,*value* . . .] [; *comments*]

Description: The .DL directive generates consecutive 32-bit integer values. The 4 bytes of the value are stored starting with the low byte of the value in low memory. Any label will be given the word type attribute. This directive may only appear in a ROM type section. The value must be a single 32-bit number. An expression is not valid.

If the program section is ROM16 and word aligned, .DL is aligned to an even address (along with label), with a zero byte in the skipped byte.

Example:

```
1.          .DL    0x12345678
2. LONG:    .DL    0xffffffff
```

Example 1 will store the four bytes 0x78, 0x56, 0x34 and 0x12 in consecutive memory locations.

Example 2 will store a 32-bit -1 in consecutive locations.

2.9.6 .do, .enddo, .exit, exitm Directives — Macro Time Looping

Syntax: [*label:*] **.DO** *count* [; *comments*]

[*label:*] **.ENDDO** [; *comments*]

[*label:*] **.EXIT** [; *comments*]

[*label:*] **.EXITM** [; *comments*]

Description: These directives are used to delimit a block of statements which are repeatedly assembled. The number of times the block will be assembled is specified by the .DO directive *count* value. The following is the format of a .DO — .ENDDO block:

Example:

```
.DO count
.
.
.
source
.
.
.
.ENDDO
```

The .EXIT directive is used to terminate a .DO — .ENDDO block before the count is exhausted. This directive allows the current pass through the loop to finish and then terminates looping. The .EXIT directive is commonly used in conjunction with a conditional test within a macro loop which will exit from the loop if a variable is equal to a particular value. In such cases the .DO *count* value is not crucial, provided it exceeds the maximum number of times the .DO loop will be required or expected to be executed for a particular macro definition or for possible macro calls.

.EXITM is similar to .EXIT except .EXIT allows the macro expansion to continue until the end of the macro is reached, while .EXITM terminates the macro expansion immediately.

Example:

```
.DO count
  .
  .
  .
.IF count
.EXIT
.ENDIF
.ENDDO
```

2.9.7 .doparm Directive — Macro Time Looping

Syntax: [*label*] **.DOPARM** *formal,'list'* [*; comments*]

Description: The .DOPARM directive repeats a macro block a number of times depending upon the number of parameters in *'list'*. During each expansion the formal parameter is replaced by the next actual parameter in the *'list'*. The *'list'* may be empty in which case one expansion will take place with a null actual parameter.

Example:

```
.DOPARM    P1,'FLAG1,FLAG2,FLAG3'
LD         P1,#0
.ENDDO
```

This will expand to:

```
LD    FLAG1,#0
LD    FLAG2,#0
LD    FLAG3,#0
```

2.9.8 .dsb, .dsw, .dsl, .dsf Directives — Define Storage

Syntax:

[*label:*] **.DSB** *size* [; *comments*]

[*label:*] **.DSW** *size* [; *comments*]

[*label:*] **.DSL** *size* [; *comments*]

[*label:*] **.DSF** *size* [; *comments*]

Description: These directives allocate a block of storage whose contents is undefined. *Size* is the size in bytes for .DSB, in words for .DSW and double words for .DSL or .DSF. *Size* must be defined during pass 1. If the program section is ROM16, RAM16 or BASE and word aligned, .DSW, .DSF or .DSL is aligned to an even address (the label is also aligned.) In a ROM section, a NOP is added when .DSW, .DSF or .DSL is aligned. A label on a .DSB is given the byte attribute. A label on a .DSW, .DSL or .DSF is given the word attribute.

Example:

```
BYT: .DSB 5     ; 5-bytes
WRD: .DSW 5     ; 5 words (10-bytes)
LNG: .DSL 2     ; 2 longs (8-bytes)
```

2.9.9 .else Directive

See .IF, .IFC, .ELSE, .ENDIF: (Conditional Assembly), Section 2.9.21.

2.9.10 .end Directive

Syntax: [*label:*] .END [*reset label*] [*; comments*]

Description: The .END directive marks the physical end of the source program. Any assembly source statement appearing after this directive is ignored. All assembly programs must use the .END directive. The optional *reset label* following .END indicates the start address for the program; the address of the label is put into the reset vector (at word location 0FFFE.) The *reset label* must be in the current module and must be of type ROM. If the program consists of multiple modules, only one should contain a reset vector.

Example:

```
START:          .
                .                   ; source code
                .                   ; START is starting address of program
          .END START                ; end of program
```

An included source file (see .INCLD directive) may optionally contain a .END directive, which is treated as an end-of-file but does not end the assembly. A *reset label* must **not** appear on the .END in an include file.

2.9.11 .enddo Directive

See .DO, .ENDDO, .EXIT Directives — Macro Time Looping, Section 2.9.6.

2.9.12 .endif Directive

See .IF, .IFC, .ELSE, .ENDIF: (Conditional Assembly), Section 2.9.21.

2.9.13 .endm Directive

Syntax: [*label:*] **.ENDM** [; *comments*]

Description: The .ENDM directive marks the end of a macro definition. All macros must end with the .ENDM directive.

Example:

```
.MACRO EXMP
 .                ; macro definition code
 source
 .
.ENDM             ; end of macro
```

The optional label is in the macro definition, but the comment is not.

See Section 2.6.1 for more examples.

2.9.14 .endsect Directive

See .SECT, .ENDSECT Directive — Program Section Directives, Section 2.9.36.

2.9.15 .error, .warning Directive

Syntax: [*label:*] **.ERROR** [*'string'*] [; *comments*]

[*label:*] **.WARNING** [*'string'*] [; *comments*]

Description: The .ERROR directive generates an error message and an assembly error that is included in the count at the end of the program. The .WARNING directive generates a warning message that is included in the warning count. These directives are useful for parameter checking.

Example:

```
.IF       VALUE<16       ; test value to see <16
LD        A,#VALUE       ; if so, generate instruction
.ELSE
.ERROR    'value>=16'    ; else generate error
.ENDIF
```

2.9.16 .even Directive

Syntax: [*label:*] **.EVEN** [*; comments*]

Description: The .EVEN directive is provided to force the location counter to the next even address, so that data following that address may be accessed as a word reference. This directive may only be used if program section is word aligned. In a ROM program section, a NOP is added when the location counter is aligned. Note that the label is aligned also.

Example:

```
.EVEN
.BYTE  1,2,3,4     ; this data may be accessed as memory words
```

2.9.17 .exit, .exitm Directive

See .DO, .ENDDO, .EXIT, .EXITM Directives — Macro Time Looping, Section 2.9.6.

2.9.18 .extrn Directive

Syntax: [*label:*] **.EXTRN** *symbol* [*:type*] [*,symbol*[*:type*]...]

Description: The .EXTRN specifies symbols that are defined public in other modules, with the .PUBLIC directive, but used in this module. The .EXTRN directive should appear in the same section as that in which the symbol was declared in the other module and be given the same byte, word or null attribute. If the symbol is an absolute, non-label value (*e.g.*, Q=3:BYTE), the .EXTRN should appear in a non-BASE section if 16-bit, else a BASE section if known to be 8-bit. Symbols in a BASE section are considered to be 8-bit values; elsewhere 16-bit values. An external specified before all section definitions is assumed 16-bit.

Example:

```
Module 1                                ; start of program
            .EXTRN VALUE:WORD           ; non-label, word attribute
            .SECT CSECT,ROM16
            .EXTRN LABEL, Q:BYTE        ; label types
Module 2                                ; start of program
            .PUBLIC VALUE, LABEL, Q
            .SECT CSECT,ROM16
            VALUE=0x222:WORD            ; non-label, word attribute
            LABEL:LD A,#10              ; label type, null attribute
                  RET
                Q:DB 10                 ; label type, byte attribute
```

The external is given the byte or word attribute by specifying :BYTE (or :B) or :WORD (or :W). If no type is specified, the size attribute is null.

The symbol relocation type is the same as the section it is defined in. Therefore, to use an external on a JSRL instruction it should be declared in a ROM type section.

NOTE: In general, it is more code efficient to place absolute value symbols (*e.g.*, Q=2:WORD or Z=3) in an include file and include the file in an assembly rather than pass the symbols externally using .EXTRN. This is because the assembler may assign a 16-bit field for an instruction with an external expression; if the expression were absolute, the assembler may determine that only an 8-bit field is required.

2.9.19 .fb, .fw Directives

Syntax: [*label:*] **.FB** *size* , *fill*

[*label:*] **.FW** *size* , *fill*

Description: These directives allocate a block of memory which is *size* of bytes or words in length. *Size* must be absolute and defined during pass 1.

Fill specifies the value to which each byte or word in the block will be set. This value may be absolute or relocatable, but may not be complex. An error is indicated if the value will not fit within a byte for .FB.

The label refers to the address of the first byte of data. For .FB, the label is assigned byte type. For .FW, the label is assigned word type and the data and label are word aligned if in ROM16, word aligned section. When alignment occurs, a zero byte is placed in the skipped byte.

.FB and .FW are valid only in ROM type section.

Examples:

```
INIT:   .FB   0x100,0     ; set 0x100 bytes to zero
TOP:    .FW   20,0xfff    ; 20 words filled with 0xfff
```

2.9.20 .form Directive

Syntax: [*label:*] **.FORM** [*'string'*][; *comments*]

Description: The .FORM directive spaces forward to the top of the next page of the output listing (it performs a form feed). The optional *string* is printed as a page title on each page until a .FORM directive containing a new *string* is encountered.

If the assembler generates a top-of-form because the listing page was full and then immediately encounters a .FORM directive, the assembler does not generate a top-of-form for the directive.

Example: .FORM 'BCD ARITHMETIC ROUTINES'

2.9.21 .if, .ifc, .ifstr, .ifb, .ifdef, .ifndef, .ifnb, .else, .endif: (Conditional Assembly) Directives

Syntax:

[*label:*] **.IF** *expression* [; *comments*]
[*label:*] **.ELSE** [; *comments*]
[*label:*] **.ENDIF** [; *comments*]

OR

[*label:*] **.IFSTR** *string1 operator string2* [; *comments*]
[*label:*] **.ELSE** [; *comments*]
[*label:*] **.ENDIF** [; *comments*]

OR

[*label:*] **.IFB** *argument* [; *comments*]
[*label:*] **.ELSE** [; *comments*]
[*label:*] **.ENDIF** [; *comments*]

OR

[*label:*] **.IFNB** *argument* [; *comments*]
[*label:*] **.ELSE** [; *comments*]
[*label:*] **.ENDIF** [; *comments*]

OR

[*label:*] **.IFDEF** *symbol* [; *comments*]
[*label:*] **.ELSE** [; *comments*]
[*label:*] **.ENDIF** [; *comments*]

OR

[*label:*] **.IFNDEF** *symbol* [; *comments*]
[*label:*] **.ELSE** [; *comments*]
[*label:*] **.ENDIF** [; *comments*]

Description: The conditional assembly directives selectively assemble portions of a source program based on the operand field of the directive statement. All source statements between a .IF, .IFC, .IFSTR, .IFB, .IFNB, .IFDEF or .IFNDEF directives and its associated .ENDIF are defined as a .IF-.ENDIF block. These blocks may be nested 99 levels deep.

The .ELSE directive can be optionally included in a .IF-.ENDIF block. The .ELSE directive divides the block into two parts. The first part of the source statements block is assembled if the .IF *expression* is not equal to zero; otherwise, the second part is assembled. When the .ELSE directive is not included in a block, the block is assembled only if the .IF *expression* is not equal to zero. If an error is detected in the *expression*, the assembler assumes a false value (zero). The *expression* must be defined during pass 1 (not containing a forward referenced value.)

The .IFSTR directive is optionally called .IFC. It allows conditional assembly based on character strings rather than the value of an expression. The *string1* and *string2* are the character strings to be compared. The *operator* is the relational operator between strings. Two operators are allowed: EQ (equal) and NE (not equal). If the relational operator is satisfied, the lines following the .IFSTR are assembled until a .ELSE or .ENDIF directive.

The primary application of the .IFSTR is to compare a macro parameter value against a specific string, for example:

```
    .IFSTR @3 NE INT     ; see if third macro argument is string "INT".
```

String1 or *string* may be enclosed within quotes, if the string contains special characters such as a blank. For example:

```
    .IFSTR 'ADD X' EQ '@1'
```

A quote within this type of string is specified by two quotes. For example:

```
    .IFSTR @2 NE 'don''t'
```

The .IFB and .IFNB directives allow conditional assembly based on whether the operand is blank (.IFB) or non-blank (.IFNB). A comment field, as the only thing following the directive, is considered a blank operand.

The .IFDEF and .IFNDEF directives allow conditional assembly based on whether the symbol is defined (.IFDEF) or undefined (.IFNDEF). The operand must consist of a single symbol and the symbol, if considered undefined, if it is a forward reference.

Listing of conditional assembly code is controlled by the .LIST directive.

Labels may optionally appear on a conditional directive line. Section 2.6.6 describes the use of conditionals in macros.

Two examples of the use of the conditional assembly directives are:

Examples: 1. Two-part conditional assembly:

```
.IF      COMPR
.                   ; Assembled if COMPR non-zero.
.
.
.ELSE
.
.                   ; Assembled if COMPR is equal to zero.
.
.ENDIF
```

2. Nested .IF-.ENDIF block:

```
.IF      SMT
.                   ; Assembled if SMT is not zero.
.
.
.ELSE
.                   ; Assembled if SMT is equal to
.                   ; zero.
.
.IF      OBR
.                   ; Assembled if OBR is not zero
.                   ; and SMT is equal to zero.
.
.ENDIF
.                   ; Assembled if SMT is
.                   ; equal to zero.
.
.ENDIF
```

2.9.22 .incld Directive

Syntax: [*label:*] .INCLD *filename* [; *comments*]

Description: The .INCLD directive includes the assembler statements in the file *filename* as part of the code being assembled.

The file must be a symbolic file; the default file extension is determined as for the source file on the invocation line.

If the included lines are to be listed, the proper control line must be in the source code. This must precede the .INCLD directive.

More exactly, the .INCLD directive causes the assembler to read source code from the specified file until an end-of-file mark (or .END directive) is reached, at which time it will again start reading source code from the assembly input file.

Example:

```
.LIST     X'21              ; expand .INCLD source code on
                            ; output listing.
.INCLD    A:BCDADD          ; include A:BCDADD.ASM file.
```

2.9.23 .ipt Directive

Syntax: [*label:*] .IPT *interrupt, addr* [; *comments*]

Description: The .IPT directive is used to create an *interrupt* vector table at word locations 0FFF0 through 0FFFC. The value of *interrupt* may be 1 through 7. The *addr* is the address to place in the table (*addr* should be defined in a ROM type section). The *interrupt* value 1 places *addr* at word location 0FFFC, the *interrupt* value 2 at location 0FFFA; down to value 7 which places *addr* at location 0FFF0. The .END directive is used to specify the reset vector at word address 0FFFE. If source consists of more than one module, only one module should contain a particular interrupt vector.

2.9.24 .list Directive

Syntax: [*label:*] **.LIST** *expression* [; *comments*]

Description: The .LIST directive controls listing of the source program. This includes listing of assembled code in general, listing of unassembled source lines contained in a .IF-.ENDIF block, and listing of code generated by the .INCLD directive, listing of macro code, and listing of warning messages.

The .LIST directive is equivalent to specifying several controls at once. The directive is maintained for compatibility with older assemblers.

Control of the various list options depends upon the seven least significant bits of the evaluated expression in the operand field (bits 6 through 0). Table 2-7 shows the options available, their associated bit weights and equivalent controls.

Options may be combined to produce the desired listing.

Examples:

1. Full Master listing:

```
.LIST 1
```

2. Suppress listing:

```
.LIST 0
```

TABLE 2-7. LIST OPTIONS

CONTROL FUNCTION	BIT		7-BIT HEX VALUE	EQUIVALENT CONTROLS
	POSITIONS	BINARY VALUE		
Master List	0	0	00	NOMASTERLIST
		1	01	MASTERLIST
.IF List	1	0	00	NOCNDLINES NOCNDDIRECTIVES
		1	02	CNDLINES CNDDIRECTIVES
.MACRO List	3,2	00	00	MCALLS NOMEXPANSIONS MOBJECT
		10	08	MCALLS MEXPANSIONS MOBJECT
		11	0C	MCALLS MEXPANSIONS NOMOBJECT
Binary List	4	0	00	NODATADIRECTIVES
		1	10	DATADIRECTIVES
Include List	5	0	00	NOILINES
		1	20	ILINES
Warning List	6	0	00	NOWARNINGS
		1	40	WARNINGS

2.9.25 .local Directive

Syntax: [*label:*] .LOCAL [*; comments*]

Description: The .LOCAL directive establishes a new program section for local labels (labels beginning with a dollar sign [$]). All local labels between two .LOCAL directive statements have their values assigned to them only within that particular section of the program. Note that a .LOCAL directive is assumed at the beginning and the end of a program; thus, one .LOCAL directive within a program divides the program into two local sections. Up to 255 .LOCAL directives may appear in one assembly.

Example:

```
$X:   .WORD 1        ; first label $X
      .LOCAL         ; establish new local symbol section
$X:   .WORD 1        ; second label $X, no confusion since
                     ; they are in different "LOCAL" blocks
```

2.9.26 .macro Directive

Syntax: [*label:*] **.MACRO** *mname* [*,parameters*] [*; comments*]

Description: The .MACRO directive names a macro and signifies the start of the macro definition.

The *mname* is the name of the macro. The name must conform to the definition of a symbol, see Section 2.3.3. The *parameters* are used in the macro definition. Each parameter must also conform to the symbol definition rules.

See Section 2.6.1, for a description of macro definition. Note that the optional label and comment on the directive line are not included in the macro definition.

2.9.27 .mdel Directive

Syntax: [*label:*] .MDEL *mname* [,*mname*]... [; *comments*]

Description: The .MDEL directive deletes the macro definition and restores the previous macro definition of the same name (if any).

Example: .MDEL INC ; INC is previously defined macro

2.9.28 .mloc Directive

Syntax: [*label:*] **.MLOC** *symbol* [,*symbol*]... [; *comments*]

Description: When a label is defined within a macro, a duplicate definition results with the second and each subsequent call of the macro. This problem can be avoided by using the .MLOC directive to declare labels local to the macro definition.

Refer to Section 2.6.5, for an example of the .MLOC directive.

2.9.29 .odd Directive

Syntax: [*label:*] **.ODD** [; *comments*]

Description: The .ODD directive is provided to force the location counter to the next odd address, (for example, so that a JIDW instruction and following table may be properly located; see .PTW directive in Section 2.9.34). The .ODD directive is only valid in a word aligned program section. If the section type is ROM, a NOP is added when the location counter is aligned. Note that the label will be aligned with the location counter.

2.9.30 .opdef Directive

Syntax: [*label*] **.OPDEF** *mname, opcode*

Description: Opdef assigns another name, *mname*, to the specified *opcode*. This name can then be used just like the original opcode. Of course, the original opcode is still available. *Opcode* may be a standard opcode, a previously defined macro, or a symbol defined in a previous .OPDEF.

Examples:

```
.OPDEF   .FLOAT,.DF     ; can now use .FLOAT same as .DF
.OPDEF   IRP,.DOPARM    ; note that even though .DOPARM is a
                        ; directive IRP doesn't necessarily
                        ; require a leading period.
```

2.9.31 .org Directive

Syntax: [*label:*] **.ORG** *expression* [; *comments*]

Description: The .ORG directive is used to set the program counter to a new value. Any bytes skipped have an indeterminate value.

The *expression* must not contain any forward references and must be either absolute or relocatable in the same section as the current program counter. The optional label is set to the value of the new program counter.

A .ORG expression is equivalent to a .= expression.

In an absolute section (see Section 2.9.36, .SECT directive), the program counter may not be set to a value lower than that specified on the .SECT directive, if any.

2.9.32 .outall, .out1, .out2, .out Directive

Syntax:

[*label:*] **.OUTALL** *'string'* [; *comments*]

[*label:*] **.OUT1** *'string'* [; *comments*]

[*label:*] **.OUT2** *'string'* [; *comments*]

[*label:*] **.OUT** *'string'* [; *comments*]

Description: These directives write a message to the console. The .OUT1 is performed only during pass 1 of the assembly, .OUT2 during pass 2 and .OUT on both passes. If the assembler is in optimize mode, then any messages output during pass 1 will be output only during the first pass 1 phase. The .OUTALL may be used to output a message on all passes of the assembler. This includes all pass 1 optimization phases and pass 2.

The operand field contains the message to be output.

Example:

```
.OUT1    'Hit end of Pass 1'
.OUT     'ARG contains bad value'
```

2.9.33 .pt Directive

Syntax: [*label:*] .PT *address* [*,address...*] [*; comments*]

Description: The .PT directive generates consecutive 8-bit bytes of data for the JID instruction. The calculation is byte=address-., where *address* is the .PT directive operand, and "." is the location of the byte. Therefore, *address* can be in the range .+0 to .+255. The .PT directive is only valid in ROM section. Each address must be defined in a ROM type section.

Example:

```
        JID
        .PT     Y,Z
Y:
        LD      A,#100
        JP      D
Z:
        LD      B,#10
D:
```

The JID instruction is used to jump to label Y or Z.

2.9.34 .ptw Directive

Syntax: [*label:*] .PTW *address* [*,address...*] [*; comments*]

Description: The .PTW directive generates consecutive 16-bit words of data for the JIDW instruction. The calculation is word=address-., where *address* is the .PTW directive operand, and "." is location of the byte. There is no range limit on the *address*. The directive must be at an even address (the immediately preceding JIDW instruction must be at an odd address; use the .ODD directive, Section 2.9.29.) This directive is only valid in a word aligned ROM16 program section. Each address must be defined in a ROM type section.

Example:

```
        .ODD                    ; force odd address
        JIDW                    ; use JIDW, Y and Z are out of JID range
        .PTW        Y,Z
        .=.+0400H               ; free area for some use
Y:
        LD          A,#100
        JP          D
Z:
        LD          B,#10
D:
```

The JIDW instruction is used to jump to label Y or Z.

2.9.35 .public Directive

Syntax: [*label:*] .PUBLIC *symbol* [,*symbol...*]

Description: This directive specifies which symbols are to be made available to other modules. In the other module, a .EXTRN directive is used. A symbol may be any null or label value.

Example:

```
.PUBLIC DOG,SYM
DOG: .DW 2
   SYM=2
```

2.9.36 .sect, .endsect Program Section Directive

Syntax: [*label*:] .SECT *name*, [*memory* [,*type* [=*addr*]] [,*align*]] [; *comments*]

[*label*:] .ENDSECT [; *comments*]

Description: This directive defines a program section. It specifies a section name and attributes. If this is the first use of the section, its location counter is set to zero. If the section has already been used, its location counter is set to its previous value. The section names and attributes are used by the linker to combine similarly named sections from other modules.

name Specifies section name. This can be any valid symbol name.

memory ROM8 is 8-bit read only memory, ROM16 is 16-bit read only memory and both hold code/constant data. RAM8 is 8-bit read write memory, RAM16 is 16-bit read write memory and both hold data. BASE is basepage RAM, which implies it lies in the address range 0-0xff and can be addressed with an 8-bit value for code efficiency.

A ROM8 or RAM8 section defaults to be aligned on a byte boundary. A ROM16 or RAM16 or BASE section defaults to be aligned on a word boundary.

type ABS is absolute. REL is relocatable (default).

For ABS, an optional address may be specified, *e.g.*, ABS=0xf000. This sets the program counter to this address and also indicates the lowest value the program counter may have in this section. The default is an address of 0.

align Specifies the boundary which the loader will place the section on. This may be

byte	any address
word	even address
long	address divisible by 4
page	address divisible by 0x100

If only the section name is given, the attributes from the first definition of that section name are used. A section name may not be defined again with different attributes.

Example:

```
.SECT CSECT,ROM8,ABS          ; define absolute byte wide ROM section
.SECT DSECT,RAM16             ; define relocatable, word wide, RAM section
.SECT CSECT                   ; same as CSECT above
.SECT A1,ROM16,ABS=0200       ; absolute code starting at 0x200
```

The .ENDSECT is optionally used to end a program section (otherwise the next without defining it again.

Example:

```
.SECT CSECT,ROM16     ; define CSECT section
.DB 2
.SECT DSECT,RAM16     ; define DSECT section
.DSB 1
.ENDSECT              ; end DSECT section
.DW 2                 ; back to CSECT section
```

NOTES:

1. A .SECT directive must appear at the start of a module before first data or code usage.
2. Data can only be word accessed in 16-bit memory (BASE, ROM16, RAM16) which is not byte-aligned. Data in 8-bit memory (ROM8, RAM8) or any byte-aligned section can only be accessed as byte data.
3. Word data is aligned to an even address. Therefore, for efficient use of data space, it is best to not mix word and byte data. For optimum usage of data space, place all word data in BASE, ROM16 or RAM16 word-aligned sections and byte data in any byte-aligned section.
4. The assembler will generate a 16-bit immediate value for any BASE page address whose offset is outside the range 0-0xff. This is the only way correct code is assured. For example:

```
        .SECT   B1,BASE
L1:     .DSB    1
L2:     .DSB    2
        .SECT   CODE,ROM16
        LD      A,#L1
        LD      A,#L1-1
        LD      A,#L2-1
```

For the first LD the offset of L1 from the start of section B1 is 0 and the immediate only requires 8 bits. The second LD has an offset of -1 (0xffff). Hence after linking, L1-1 could actually have the value -1, thus the immediate will require 16 bits. The final LD has an offset of 0, and hence only requires 8 bits. Note that the user could always force an 8-bit immediate by using the low operator, *e.g.*, LD A,#LOW L1-1.

2.9.37 .set Directive

Syntax: [*label:*] **.SET** *symbol, expression* [*:type*][; *comments*]

Description: The .SET directive is used to assign values to *symbols*. In contrast to an assignment statement, a *symbol* assigned a value with the .SET directive can be assigned a new value at any place within an assembly language program.

The optional type may be specified as byte (:BYTE or :B) or word (:WORD or :W) and will assign this type to the symbol. Use of type will override the type of the expression.

Example:

```
.SET    A1,0                ; set A1=0
.SET    A1,100              ; set A1=100
.SET    B1,50               ; set B1=50
.SET    C1,A1-25*B1/4       ; set C1=A1-25*B1/4
.SET    D1,A1:BYTE          ; set D1= to value 100 with byte type
```

2.9.38 .space Directive

Syntax: [*label:*] .SPACE *expression* [; *comments*]

Description: The .SPACE directive inserts a number of blank lines into the output listing. The number of lines inserted is specified by the *expression* in the operand field.

Example:

```
.SPACE 20                ; skip 20 lines.
```

2.9.39 .spt Directive

Syntax: [*label*:] **.SPT** *subroutine label* [; *comments*]

Description: The .SPT directive generates one of the subroutine addresses to be placed in the 16 word ROM table at address 0FFD0 through 0FFEE hex. Up to 16 .SPT directives may be specified. Each subroutine label must specify a ROM type label. The label may be an external symbol.

NOTE: The assembler merely passes information to the linker to generate the ROM table. The assembler does not actually store anything in locations 0FFD0 through 0FFEE.

This ROM table is used to provide single byte subroutine calls for up to 16 subroutine labels. The .SPT directive will automatically convert JSR's and JSRL's with one of these labels to a single byte instruction, if the following conditions are met:

1. Bit 0 of the .CONTRL directive is 1 to allow code optimization (this is the default value, see Section 2.9.3.)
2. In two pass mode, the .SPT directive appears before the JSR or JSRL instruction which uses that subroutine label as an operand. In optimize mode the restriction does not apply.
3. The label operand of the .SPT directive is exactly the same as the label operand of the JSR or JSRL instruction (that is, the symbolic name is the same). This is because the assembler must determine if the JSR or JSRL is a single byte by its symbolic label, and not by its operand address.

Example:

```
                        ; assume default .CONTRL value
      .SPT    L1
L1:                     ; subroutine
L2:   NOP
      RET
      JSR     L1        ; JSR reduced to single byte opcode
      JSRL    L1        ; JSRL reduced to single byte opcode
      JSR     L2        ; not reduced, different label name
      JSR     L1+0      ; not reduced, operand is not label
```

Example:

```
      .SPT          L1
L1:
      RET
      .CONTRL 0
      JSR           L1    ; not reduced to single byte because of .CONTRL
```

If there are multiple source modules, all the .SPT directives should be declared with the appropriate .PUBLIC or .EXTRN directives.

Example: source module 1

```
  .SECT CODE,ROM16
  .SPT L3
  .SPT L2
  .PUBLIC L2
  .EXTRN L3
  JSR L2                     ; single byte instruction
  JSR L3                     ; single byte instruction
L2:
```

source module 2

```
  .SECT CODE,ROM16
  .SPT L3
  .SPT L2
  .SPT L1
  .PUBLIC L1,L3
  .EXTRN L2
  JSR L1                     ; single byte instruction
  JSR L2                     ; single byte instruction
L1:
L3:
```

2.9.40 .title Directive

Syntax: [*label:*] **.TITLE** [*symbol*][,'*string*'][; *comments*]

Description: The .TITLE directive identifies the output listing in which it appears with an optional *symbolic name* and an optional title *string*. The symbolic name is also used as the module name in the object module. This is the name that will appear in the linker load map. (If the symbol is not specified the module name is formed from the source filename.) If more than one .TITLE directive is used, the last one encountered specifies the *symbolic name* and *string*. The *symbolic name* must meet the symbol construction restrictions given in Section 2.3.3. Single quotes (') must appear at the beginning and end of the character *string*. A single quote mark in the string is represented by two quote marks (").

Example: .TITLE TBLKP, 'TABLE LOOKUP'

prints

```
TBLKP    TABLE LOOKUP
```

on the output listing and module name is TBLKP.

```
.TITLE   ,'DATA TABLE'  ; use default module name.
```

2.9.41 .word, .dw Directives

Syntax: [*label:*] **.WORD** *expression* [*,expression...*] [; *comments*]

[*label:*] **.DW** *expression* [*,expression...*] [; *comments*]

Description: The .WORD and .DW directive generates consecutive 16-bit words of data for each given expression. If the directive has a label, it refers to the address of the first word. If the program section is ROM16 and word aligned, .WORD (and .DW) is aligned to an even address (along with *label.*) If alignment occurs, a zero byte is placed in the skipped byte. The program section must be ROM type. Any label will be assigned the word type.

Examples:

```
1.      .WORD  0FF
2. T:   .DW  MPR-10,0FFFF,'AB'
```

Example 1 stores the hexadecimal number FF in a word of memory.

Example 2 stores three 16-bit values in consecutive words in memory.

To store ASCII strings in consecutive bytes, use the .BYTE directive, Section 2.9.1.

2.10 ASSEMBLER CONTROLS

This section describes the controls that may be used in the source program on a control line or used on the invocation line as an option. The syntax of their usage as an option is described in Section 2.2.2.

A control line is indicated by a # in column 1 of the source line followed by any of the following controls separated by white space. Comments may be included on a control line by using a semicolon followed by the comment. The ";" terminates the control line.

A control name may be abbreviated to the number of letters shown in capital letters in the descriptions. For example, MASTERLIST may be also specified as MASTER, MA, or even M. However, MC would give an error as we can't tell if MCALLS or MCOMMENTS is wanted. Many of the controls may also have the prefix "NO". In this case, the rest of the name may be abbreviated as normal. Control names are not case sensitive.

A control may be classified as primary or general or invocation line only. Primary controls are those that can only be set on the invocation line or at the beginning of the program before any other statements except for other control lines or comments. General controls may be specified anywhere and can be respecified at any time. Thus, the usual source program structure would be:

```
#MASTERLIST              ;general is okay up here
; set up for debugging
#DEBUG                   ;but primary must be here
; maybe a few comments
        NOP
; at this point only general controls will be valid
```

A few of the controls are shown as "invocation line only." This, of course, means they are only valid on the program invocation line and cannot be used within the program.

A general control may be used by the SAVE and RESTORE controls.

Invocation line controls are masters and override the same control in the program source. Thus, NOMASTERLIST on the invocation line will override #MASTERLIST in source.

Assembler controls and the functions, described in this section, are summarized in Table 2-8.

TABLE 2-8. SUMMARY OF ASSEMBLER CONTROLS

CONTROL	FUNCTION	SECTION
[NO]Aserrorfile[=*file*]	[NO]Create a source file with errors	2.10.1
[NO]CNDDirectives	[NO]List of conditional directives	2.10.2
[NO]CNDLines	[NO]List lines of conditional code	2.10.3
[NO]COMMentlines	[NO]List comment lines	2.10.4
[NO]COMPlexrel	[NO]Complex relocation	2.10.5
[NO]CONstants	[NO]List constants in symbol table	2.10.6
[NO]Crossref	[NO]Cross-reference table in listing file	2.10.7
[NO]DAtadirectives	[NO]List all lines of object code	2.10.8
Define *symbol*[=*value*]	Define a symbol	2.10.9
[NO]ECho	[NO]Echo command file to the console.	2.10.10
[NO]Errorfile[=*file*]	[NO]Error file	2.10.11
[NO]Formfeed	[NO]Form feeds	2.10.12
[NO]Headings	[NO]Heading on each list page	2.10.13
[NO]ILines	[NO]List include file	2.10.14
Include=*directory*	Additional include search directory	2.10.15
[NO]Listfile[=*file*]	[NO]List file	2.10.16
[NO]LOcalsymbols	[NO]Local symbols in object file	2.10.17
[NO]Masterlist	[NO]List of source lines in list file	2.10.18
[NO]MCAlls	[NO]List of macro call statements	2.10.19
[NO]MComments	[NO]Macro comments saved in definition	2.10.20
[NO]MDefinitions	[NO]List of macro definition	2.10.21

TABLE 2-8. (Cont)

CONTROL	FUNCTION	SECTION
[NO]MExpansions	[NO]List macro expansion lines	2.10.22
[NO]MLocal	[NO]Macro local symbols in symbol table	2.10.23
[NO]MObject	[NO]List macro object lines only	2.10.24
[NO]Numberlines	[NO]Number list lines by source file	2.10.25
[NO]Objectfile[=*file*]	[NO]Object file	2.10.26
PAss=*number*	Number of passes assembler performs	2.10.27
PLength=*number*	Number of lines for a page	2.10.28
PWidth=*number*	Number of characters per line	2.10.29
Quick	Fast list only	2.10.30
[NO]Remove	[NO]Remove source file error lines	2.10.31
REStore	Restore state of controls that were saved	2.10.32
SAve	Saves state of general controls	2.10.33
[NO]SIGnedcompare	[NO]Comparisons using signed arithmetic	2.10.34
SIZesymbol=*number*	Specifies maximum symbol size	2.10.35
[NO]Tablesymbols	[NO]Symbol table in list file	2.10.36
[NO]TABS	[NO]Tabs in list file	2.10.37
Undefine *symbol*	Undefine a symbol	2.10.38
[NO]UPpercase	[NO]Convert symbols to uppercase	2.10.39
[NO]Verify	[NO]List all passes of assembler	2.10.40
[NO]Warnings	[NO]List warning messages	2.10.41
[NO]Xdirectory	[NO]Search only specified Include directories	2.10.42

2.10.1 Aserrorfile

Syntax: [NO]**Aserrorfile** [=*filename*]

Description: Causes the program to create a source error file. This file will contain all the lines that were in the source program plus error messages for those lines containing errors. The user can then edit this file and quickly fix the errors, as both the source statements and errors are combined. In the typical case, the user would then run this file through the assembler using the REMOVE control with ASERRORFILE specified again. An error free source file can then be created. The class is invocation line only. The default is NOASERRORFILE. The default filename extension is .ase.

Example:

```
A                        ; uses file sourcefile.ase
ASE=MAIN.NEW
```

2.10.2 Cnddirectives

Syntax: **[NO]CNDDirectives**

Description: This control enables/disables the listing of conditional directives (*e.g.*, .IF, .ELSE, .ENDIF). It is overridden by the NOCNDLINES control. Thus, while NOCNDDIRECTIVES can be used to disable listing all conditional directives, CNDD will not list those lines suppressed by NOCNDLINES. The default is NOCNDDIRECTIVES. The class is general.

2.10.3 Cndlines

Syntax: **[NO]CNDLines**

Description: CNDLINES causes lines of code that are not assembled, because of conditional assembly, to be listed. NOCNDLINES inhibits the listing of these lines. The default is NOCNDLINES. The class is general.

2.10.4 Commentlines

Syntax: **[NO]COMMentlines**

Description: This control allows the user to have comment lines listed or not listed in the output listing. It is provided to allow the user to get a quick listing. Note that a blank line is considered a comment. The default is COMMENTLINES. The class is general.

2.10.5 Complexrel

Syntax: **[NO]COMPlexrel**

Description: This control enables/disables complex relocation. In most cases, the user isn't concerned about this control. However, in programs in which you know there are no complex expressions, it is useful, for error checking purposes to use NOCOMPLEXREL. It is the unusual program that requires complex expressions. The default is COMPLEXREL. The class is general.

2.10.6 Constants

Syntax: **[NO]CONstants**

Description: Many programs are written in which the user has hundreds of constants defined (*e.g.*, CR=13) and typically a .INCLD is used to read these into files. Normally all symbols, including these, will be listed on a symbol or cross reference table. NOCONSTANTS can be used to ignore these absolute numbers in the symbol/cross reference table, effectively "cleaning up" the listing. If LOCALSYMBOLS is effect, NOCONSTANTS also inhibits absolute numbers from being placed into the object module. Note that an absolute number is different from a label defined in an absolute section. The default is CONSTANTS. The class is primary.

2.10.7 Crossref

Syntax: **[NO]Crossref**

Description: CROSSREF causes a cross reference table to be included in the output listing. CROSSREF overrides TABLESYMBOLS. The default is NOCROSSREF. The class is primary.

2.10.8 DAtadirectives

Syntax: **[NO]DAtadirectives**

Description: Some data generation directives may display more than one line of object code in the listing file. DATADIRECTIVES allows all these lines to be listed while NODATADIRECTIVES causes only the first line to be listed. The default is DATADIRECTIVES. The class is general.

2.10.9 Define

Syntax: **Define** *symbol* [=*value*]

Description: This control is used to define a symbol from the invocation line. This could then be used within the program to create different versions, etc. The conditional assembly directive, .IFDEF or .IFNDEF, can also detect if the symbol has been defined. If no value is specified for the symbol, it will be assigned a value of 1. The class is invocation line only.

Example:

```
DEF LOOPCOUNT=10    ; set LOOPCOUNT to 10
D VERSION2          ; set VERSION2 to default of 1
```

2.10.10 Echo

Syntax: [NO]**ECho**

Description: This control specifies whether command file (*i.e.*, @files) lines will be echoed to the console. The default is NOECHO. The class is invocation line only.

2.10.11 Errorfile

Syntax: [NO]**Errorfile** [=*filename*]

Description: This control specifies the name of the file to which only errors will be written. If specified without a filename, errors will go to the console (but not if the normal listing file is also the console). NOERRORFILE indicates that no error file is to be used. The default is ERRORFILE to the console. The default filename extension is .err. The class is invocation line only.

Example:

```
ERR=MAIN     ; errors to MAIN.err
```

2.10.12 Formfeed

Syntax: **[NO]Formfeed**

Description: FORMFEED causes a form feed character to be used between pages. NOFORMFEED uses blank lines to move between pages. The number of blank lines is dependent upon the page length. If NOHEADINGS is in effect, then this control has no effect. The default is FORMFEED. The class is primary.

2.10.13 Headings

Syntax: [NO]**Headings**

Description: Normally the output listing is broken up into pages whose size is specified by the PLENGTH control. Each page is separated by a form feed or by the appropriate number of line feeds. The top of each page contains a header which contains the page number and may also have a title and date. NOHEADING indicates that there will be no page breaks in the output listing. The PLENGTH control is ignored, a single header will appear at the beginning of the listing and all other lines will be listed continuously. The default is HEADINGS. The class is primary.

2.10.14 Ilines

Syntax: **[NO]ILines**

Description: This control enables/disables the listing of lines read from .INCLD files. The default is NOILINES. The class is general.

2.10.15 Include

Syntax: **Include** = *directory*

Description: Normally, when using the .INCLD directive, the program looks in the current and default directories to find a file without an explicit directory name. If not found, it flags an error. This control enables the program to search other devices and/or directories to find the file. Multiple directories may be searched, by specifying each on a separate INCLUDE control. The class is invocation line.

Example:

```
I=C:\    ; check root directory on drive C:
I=..\    ; check parent directory
```

2.10.16 Listfile

Syntax: **[NO]Listfile [**=*filename* **]**

Description: LISTFILE causes the program to write the listing to the file specified. LISTFILE with no filename will use the default extension appended to the source filename. NOLISTFILE will inhibit any listing from being created. In this case, the default is that all errors will be displayed on the console. (See Section 2.10.11, ERRORFILE.) The default is LISTFILE with only error lines appearing in the file. If LISTFILE is specified, the file will contain all lines not inhibited by other options. The default filename extension is .lis. The class is invocation line only.

Example:

```
LIST=N     ; creates file n.lis
L=CON      ; output to the console
```

2.10.17 Localsymbols

Syntax: [NO]LOcalsymbols

Description: LOCALSYMBOLS causes all symbols to be put into the object module and thus be made available for the linker cross reference. If LOCALSYMBOLS is not used during a relocatable assembly, only those symbols declared PUBLIC will be available for the linker cross reference. The default is NOLOCALSYMBOLS. The class is primary.

2.10.18 Masterlist

Syntax: **[NO]Masterlist**

Description: This control enables/disables the listing of the source lines in the list file. NOMASTERLIST will also cause the source control line to not be listed. Lines containing errors will always be placed into the listing file. NOMASTERLIST overrides all other list directives. The default is MASTERLIST. The class is general.

2.10.19 Mcalls — Macro Calls

Syntax: [NO]MCAlls

Description: MCALLS allows the listing of macro call statements while NOMCALLS inhibits them. The default is MCALLS. The class is general.

2.10.20 Mcomments — Macro Comments

Syntax: **[NO]MCOmments**

Description: NOMCOMMENTS causes all comments within a macro definition to not be included within the definition. Thus, when this macro is expanded, the comments will not appear. Of course these comments will appear on the output listing as part of the macro definition. A macro double comment, *e.g.*, ;;COMMENT, will never appear in a macro expansion. This control allows all comments to be removed. Blank lines are considered comments. The default is MCOMMENTS. The class is general.

2.10.21 Mdefinitions

Syntax: [NO]**MDefinitions**

Description: MDEFINITIONS allows the listing of a macro definition while NOMDEFINITION inhibits it. A macro definition consists of all lines from a .MACRO, .DOPARM, or .DO directive to its matching .ENDM or .ENDDO. The default is MDEFINITIONS. The class is general.

2.10.22 Mexpansions

Syntax: **[NO]MExpansions**

Description: MEXPANSIONS causes all macro expansion lines to be listed. NOMEXPANSIONS suppresses the macro expansion lines. A macro expansion consists of those lines that are generated by a macro call or a .DO directive. The macro call line is listed in either case. The default is NOMEXPANSIONS. The class is general.

2.10.23 Mlocal — Macro Local Symbols

Syntax: **[NO]MLocal**

Description: MLOCAL causes "local macro" symbols to be included in a symbol or cross reference table. Additionally, these symbols will be placed into the object module if LOCALSYMBOLS is in effect. The default is NOMLOCAL. The class is primary.

2.10.24 Mobject — Macro Object

Syntax: [NO]MObject

Description: NOMOBJECT allows all lines of a macro expansion to be listed. MOBJECT only lists those macro lines that generate object code. This is very useful for those macros which contain a lot of conditional statements but only a few instructions. This control is overridden by NOMEXPANSIONS. The default is NOMOBJECT. The class is general.

2.10.25 Numberlines

Syntax: [NO]**Numberlines**

Description: NONUMBERLINES causes each line in the output listing to have the next sequential line number. However, because of macros or .INCLD files the line number may not reflect the actual position in the source file. While this is useful for a cross reference table, it may not be for other purposes. NUMBERLINES causes each source line to have the same line number as its position in the file. In this case, lines from macro expansions or .INCLD files will not have line numbers. The default is NUMBERLINES. The class is primary.

2.10.26 Objectfile

Syntax: [NO]**Objectfile** [=*filename*]

Description: OBJECTFILE causes the program to write the object module to the file specified. OBJECTFILE with no *filename* will use the default extension appended to the source filename. NOOBJECTFILE will inhibit any object module from being created. The default is OBJECTFILE. The default filename extension is .obj. The class is invocation line only.

Example:

```
O=DISPLAY     ; uses file display.obj
NOOBJ         ; no object module
```

2.10.27 Pass

Syntax: **PAss** = $\{$*number* | ALL$\}$

Description: PASS specifies the number of passes through the source code that the assembler should perform. An argument of ALL causes the assembler to perform the minimum number of passes necessary to optimize the size of the code. A count of 0, 1 or 2 causes the assembler to perform a standard two pass assembly. Any other value will cause the assembler to perform that number of passes. The default is ALL. The class is invocation line only.

2.10.28 Plength

Syntax: **PLength** = *number*

Description: Specifies the length in lines of each page in the listing file. This is the physical length of the page, not the number of lines that will appear on the page. Thus a printer using 8 lines/inch would specify PLENGTH=88 using standard 11 inch paper. Values less than 11 use the value 11 and values greater than 255 use 255. The default is 66. The class is primary.

PLENGTH is overridden by the NOHEADINGS control.

Example: PL=60

2.10.29 Pwidth

Syntax: **PWidth** = *number*

Description: This control sets the number of characters that will be printed on the output listing. Characters past this position are ignored. This is also used as the position to which the page number and date are aligned in the header. Values less than 64 use 64 and values greater than 128 use 128. Although some terminals are 80 characters wide, writing a CR-LF in the 80th spot causes a subsequent blank line to appear. In this case, the user should use 79. This is the case on the IBM PC. The default is 79. The class is primary.

Example: PW=100

2.10.30 Quick

Syntax: **Quick**

Description: Specify QUICK to get a fast, errors only assembly. QUICK causes NOMASTERLIST, NOLISTFILE, NOOBJECTFILE, NOASERRORFILE, NOTABLESYMBOLS, and NOCROSSREF to be in effect. It also enables ERRORFILE, which defaults to the console. If any of the above controls are used after QUICK, they can be turned back on. In most cases, the user can just enter the source file to assemble followed by only the QUICK control. The class is invocation line only.

Example:

```
Q               ; just errors to console
QUICK OBJ       ; also get object file
```

2.10.31 Remove

Syntax: **[NO]Remove**

Description: When a source error file is created by ASERRORFILE control, all errors are flagged with an "error string" at the beginning of the error message line(s). If this file is subsequently processed by the assembler, REMOVE can be used to remove these lines from both the output listing as well as another source error file. The assumption being, the user has fixed the lines in error. Of course, if there is still an error, it will appear on an error line. The default is NOREMOVE. The class is primary.

2.10.32 Restore

Syntax: **RESTore**

Description: This control restores the state of those controls which were saved with the SAVE control. An error is flagged, if RESTORE is used without a previous SAVE. The class is general.

2.10.33 Save

Syntax: **SAve**

Description: This control saves the state of general controls, except for SAVE and RESTORE themselves. SAVE's can be nested to 8 levels. The state of these controls can be restored with the RESTORE control. The typical use is to save the state of the MASTERLIST flag before calling a macro which will turn the listing off. After the macro, RESTORE is used to set the MASTERLIST flag to its starting state. If SAVE/RESTORE were not available, the user wouldn't know whether to turn the listing on or off after the macro that is not listed. The class is general.

2.10.34 Signedcompare

Syntax: **[NO]SIGnedcompare**

Description: NOSIGNEDCOMPARE causes all expressions containing conditional operators (*e.g.*, GT, GE, LT, LE) to be evaluated using unsigned arithmetic. SIGNEDCOMPARE evaluates these expressions using signed arithmetic. The default is NOSIGNEDCOMPARE. The class is general.

2.10.35 Sizesymbol

Syntax: **SIZesymbol** = *number* (6-64)

Description: Specifies the maximum symbol size. All symbols are stored as variable length so there is usually no need to change the maximum symbol size. However in certain cases, such as if the symbols will be passed to another program, there may be a need to limit symbols to a maximum size. In this case, only the maximum number of characters will be used to represent the symbol, the rest will be ignored. The default is 64. The class is primary.

2.10.36 Tablesymbols

Syntax: **[NO]Tablesymbols**

Description: NOTABLESYMBOLS specifies that no symbol table will be listed, while TABLESYMBOLS creates a symbol table listing. This control is overridden by the CROSSREF control. Note that NOMASTERLIST does not override TABLESYMBOLS. The default is NOTABLESYMBOLS. The class is primary.

2.10.37 Tabs

Syntax: **[NO]TABS**

Description: TABS specifies that tabs should be retained in the output listing while NOTABS causes tabs to be expanded into the appropriate number of blanks. Tab stops are assumed to be every 8 columns. TABS is useful in reducing the size of the output listing and possibly printing faster. The default is NOTABS. The class is general.

2.10.38 Undefine

Syntax: **Undefine** *symbol*

Description: UNDEFINE is used to undefine a symbol that was defined by the DEFINE control. Note that it can not be used to undefine a symbol defined in the source program. The class is invocation line only.

Example: U COUNT

2.10.39 Uppercase

Syntax: **[NO]UPpercase**

Description: UPPERCASE causes all lower case characters to be converted to upper case in symbols and opcodes. This does not affect characters used in strings. NOUPPERCASE allows symbols using upper and lower case to be considered differently. For example, ABCD and abcd are different symbols if NOUPPERCASE is in effect. All assembler keywords and opcodes can be any combination of upper and lower case. Thus NOP, nop, NoP are all recognized as an opcode regardless of the state of this control. The default is NOUPPERCASE. The class is primary.

2.10.38 Undefine

Syntax: **Undefine** *symbol*

Description: UNDEFINE is used to undefine a symbol that was defined by the DEFINE control. Note that it can not be used to undefine a symbol defined in the source program. The class is invocation line only.

Example: U COUNT

2.10.39 Uppercase

Syntax: **[NO]UPpercase**

Description: UPPERCASE causes all lower case characters to be converted to upper case in symbols and opcodes. This does not affect characters used in strings. NOUPPERCASE allows symbols using upper and lower case to be considered differently. For example, ABCD and abcd are different symbols if NOUPPERCASE is in effect. All assembler keywords and opcodes can be any combination of upper and lower case. Thus NOP, nop, NoP are all recognized as an opcode regardless of the state of this control. The default is NOUPPERCASE. The class is primary.

2.10.40 Verify

Syntax: **[NO]Verify**

Description: VERIFY causes an output listing to be generated during each pass of the assembler. This is mainly used to examine the effect of multi-pass optimization and typically would not be used. The default is NOVERIFY. The class is primary.

2.10.41 Warnings

Syntax: **[NO]Warnings**

Description: WARNINGS causes any warning messages to be included in the output listing. NOWARNINGS inhibits the messages. The default is WARNINGS. The class is general.

2.10.42 Xdirectory

Syntax: [NO]**Xdirectory**

Description: Normally when searching for an include file, the current, default, and any directories specified via the INCLUDE control are searched. XDIRECTORY causes only those directories specified by the INCLUDE control to be searched. This allows files in another directory that have the same name as those in the current directory to be accessed. The default is NOXDIRECTORY. The class is invocation line only.

See Section 3.6 for all linker commands.

3.2.3 Default File Names and Extension

For those options that require a filename, the name may include a directory path. It may also consist of just a directory path. In this case, the default filename will be used but written to the directory. The default filename is the name of the first object file specified, with any extension removed.

For MS-DOS systems a default extension is always placed on a file unless one is explicitly specified.

If an output filename consists of just a directory, it should always be terminated by a "\" in MS-DOS systems. If not, it will be treated as a filename. Thus /M=txt\ would output the map to file *txt\cat.map* (assuming *cat.obj* is the input file). /M=txt would output the listing to *txt.map*.

3.2.4 Library File Search Order

When searching for a file specified by the /LIBFILE option, directories are searched in the following order:

1. current directory
2. directories specified by the /LIBDIRECTORY option
3. default directories
    a. directory specified by environment variable LNHPC, if it exists
    b. directory specified by environment variable HPC, if it exists
    c. directory \HPC

If the /X option is specified, only directories in category 2 above will be used. Any filename which contains an explicit directory will only be checked for in that directory. No other directories will be searched.

3.2.5 Help File Search Order

When searching for the help file, *LNHPC.HLP* directories are searched in the following order:

1. current directory
2. default directories (as noted in Section 3.2.4).

options	is a list of linker commands described in Section 3.2.2. No options need be specified in which case the linker defaults will be used.
cmd file	is the name of a file that contains additional invocation line object filenames and/or options. This file has a default extension of .CMD for an MS-DOS system. For example, if the file *a.cmd* contains:

```
test
/table
```

and file *b.cmd* contains

```
/o=testdata
```

then the command

LNHPC @a /d @b

is equivalent to

```
LNHPC test /table /d /o=testdata
```

Any errors detected on the invocation line causes the linker to stop execution and display the part in error with an appropriate message.

Some sample invocation lines for an MS-DOS system are:

```
LNHPC  TEST  /NOOUTPUTFILE /TABLE /MAP=CON
LNHPC  MOD1,MOD2  /CROSS /BRIEF /MAP
LNHPC  \DEMO\SAMPLE /SECT data=0x400
```

The first command line links the file *test.obj* and outputs the load map to the console. No output object file is produced but a symbol table is listed.

The second command line links the object modules *mod1.obj* and *mod2.obj* and places the output object file into *mod1.hex*. A cross reference table is produced and the brief form of the map is used. The map is written to file *mod1.map*.

The next example links the file *sample.obj*, which resides in directory *demo*. It produces *sample.hex*, which will be in the current directory. A section, called data, in the object module is assigned a starting address of 0x400.

3.2.2 Linker Options

An option is a linker command specified on the invocation line in a manner consistent with the operating system.

The linker invocation line options for MS-DOS systems start with a / which may be preceded and followed by white space. Linker options are not case sensitive and may be abbreviated to the minimum number of characters as specified in the command descriptions. For example:

```
/CROSSREF / MAP
```

```
RAM16
 [size = 0000]

RAM8
 [size = 0000]

ROM16
 0200  0206
 02B0  02B7
 [size = 000F]

ROM8
 [size = 0000]

VECTOR
 FFFE   FFFF
 [size = 0002]

--Total Memory Map--

TOTAL RAM = BASE + RAM16 + RAM8
 [size = 0000]

TOTAL ROM = ROM16 + ROM8 + VECTOR
 0200  0206
 02B0  02B7
 FFFE  FFFF
 [size = 0011]

-- Section Table --

start    end       attributes      Section
                                     Module


0200     0206    ROM16 WORD      TWO
0200     0203                      SAMPLE1
0204     0206                      SAMPLE2
02B0     02B7    ROM16 WORD      ONE
02B0     02B7                      SAMPLE2
```

3.3 LINKER EXAMPLE

The following is a very simple example. Two modules were assembled and their object module processed by the linker. First the two source programs are shown followed by the linker commands to process them, followed by the linker outputs.

```
Assembly file 1. LE1.ASM      Assembly file 2. LE2.ASM

       .TITLE SAMPLE1               .TITLE SAMPLE2
       .PUBLIC P1                   .PUBLIC E1
       .SECT TWO,ROM16              .SECT ONE,ROM16
       .EXTRN  E1                   .DB   1,2,3,4
        NOP                         .DB   -1,-2,-3,-4
P1:                                 .SECT TWO,ROM16
       LD A,#E1                     .EXTRN P1
       .END                  E1:
                                    JSR P1
                                    .END E1
```

File 1 has a module name of SAMPLE1 while file 2 has one of SAMPLE2. It is important to give each assembly module a different name since this is used in the linker load map. However, it is not a requirement, as the default is to use the filename.

NOTE: SAMPLE1 has defined a section called TWO and SAMPLE2 has a section called TWO as well as a section ONE. Note that section names are not case sensitive.

After being assembled, the following invocation line was used with the linker:

```
LNHPC le1,le2 /sect one=0x2b0/sect two=0x200/out=sample/nobrief /map
```

The following is the load map output by the linker:

```
Reset Vector:   0204

-- Memory Order Map --

0200  0206   ROM16
02B0  02B7   ROM16
FFFE  FFFF   VECTOR

-- Memory Type Map --

BASE
 [size = 0000]
```

3.5 LINKER ERRORS

Table 3-1 is a list of the loader messages. These are divided into command line errors, link errors, link warnings and object module errors. Object module errors are caused by a bad object file; user should check filename and/or assemble and link again.

TABLE 3-1. LINKER ERRORS

ERROR	**DESCRIPTION**
Command Errors	
Invalid Command	the command specified is not a valid linker command.
Ambiguous Command	this command abbreviation can be more than one command.
File Not Found	the object or library file on the invocation line cannot be found. Maybe wrong extension is assumed or it's in a different directory.
Invalid File Name	the file specified is not a valid filename.
Invalid Section Name	the section name on the SECT command is not valid.
Invalid Numeric	the number contains a digit not valid for the radix.
Section Already Loaded	you must specify the section address before it is loaded.
Invalid Character	a character used in this command is invalid.
Missing Operand	operand not found.
Invalid or missing Memory Type	memory type must be ROM16, ROM8, RAM16, RAM8, or BASE.
Invalid or missing range	low address of range > high address.
Invalid Object Type or Option	invalid FORMAT option.
"NO" not valid for command	command can't use NO.
No Object file specified	No file found.
Link Errors	
Duplicate PUBLIC Symbol	the symbol shown has already been defined as a PUBLIC in a previous object module.
Word Address is not Even	the address in the specified module should be even but is odd.
Base Page Exceeds Range	modules that are declared BASE sections have exceeded the maximum address of 0xBF.

3.4 MEMORY ALLOCATION

The Linker places each section in memory based on the attributes of the section and the memory that is available, which is specified by the RANGE command. Each section has the following attributes:

memory type	— BASE, ROM8, ROM16, RAM8, RAM16
size	— determined from object modules
absolute	— section was specified as absolute in assembler
fixed	— starting address was specified by the SECT command
ranged	— memory range was specified by the SECT command

Memory is allocated section by section. Typically, there is no way to control the order in which sections will be allocated. However, sections are processed in the order in which they become known to the linker. A section is known when it appears in a SECT command or is processed in an object module.

Normally, as each section is processed, the ranges for its memory type are examined to find enough free space to allocate the section. Each range is examined in order. The first space large enough to contain the section is used. At this point, the memory allocated is marked used. If not enough memory is available to allocate the section, an error message is displayed.

Sections are allocated in the following order:

1. Each absolute or fixed section is placed in memory at its specified address. This is the case regardless of whether this memory has been allocated by a RANGE command.

2. Each ranged section is placed in memory within the specified range, regardless of whether this memory has been allocated by a RANGE command. Of course, an error will occur if the section cannot be located.

3. All remaining sections are allocated in the normal fashion. For efficiency, sections which may contain word aligned data (ROM16, RAM16, BASE which are word aligned) are allocated first. The user will benefit if the word aligned data is placed in these sections and byte data in other sections.

The Memory Order Map shows the starting and ending addresses of each contiguous range of memory. It also indicates the type of memory.

The Memory Type Map shows how memory is allocated but is organized by the type of memory. Within each type the allocation is shown in memory order.

The Total Memory Map shows allocation of all ROM and all RAM.

The section table, which is listed by the /NOBRIEF option, shows each section in the link, along with its starting and ending address. The section attributes are also displayed. The modules that comprise each section are displayed underneath the section names along with its addresses.

If there is an overlap between sections this would be indicated in the Memory Order Map by the message "** overlap **" next to the overlapping section(s).

TABLE 3-1. (Cont)

ERROR	DESCRIPTION
Object Module Errors (cont)	
Record Disp out of Range	a relocation command has a displacement that is out of range.
Invalid Relocation CMD	an object module relocation command does not have a valid value.
Missing Module Header	the first record of the object module is not valid. Perhaps the wrong file was read.
Multiple Headers	the object module contains two header records. Must be a corrupted file.
SYNC Error	something happened to this file or the system between pass 1 and pass 2 of the loader. Try to load program again.
Bad Section ID	a section ID in the object module is out of range.
Record out of order	a record in the object module is in the wrong place.
Checksum Error	an object record has a checksum error. Reassemble the file.
Invalid Record Type	an object module is invalid.
Read Past Record	the record length didn't correspond with the information record.
Bad Object Symbol	a symbol in the object module is invalid.

3.6 COMMANDS

This section describes the linker commands, giving the command name, arguments, and default state. Table 3-2 is a summary of the linker commands discussed in this section.

TABLE 3-1. (Cont)

ERROR	DESCRIPTION
Link Errors (cont)	
No End Address has been Specified	no input module has an end address.
Section Mismatch	section with the same name from two different object modules contains mismatched attributes. Reassemble one of the modules after changing the attributes or rename the section.
Word reference to ROM8/RAM8 section	cannot reference 8-bit memory as word.
Base page reference not on basepage	result of basepage expression >0FF Hex.
Undefined external	no public definition for external symbol.
Undefined Section	the specified section was used on a SECT command but never appeared in an object module. Probably a misspelled name.
No Ranges Left For Section	there is not enough memory available via the RANGE command to allocate all object modules.
Object code placed in Internal RAM area	code at <0x200 hex address.
Multiple End Addresses have been specified	only 1 end address allowed.
RAM section between 0xC0 - 0x1BF	RAM in on-chip register area.
Bad Library File	bad filename or file corrupted.
.SPT Table Full	more than 16 .SPT's specified.
Link Warnings	
Public, external byte,word type mismatch	external does not match public in byte,word type.
BASE public used as non-BASE external	public symbol declared in BASE section, not used as BASE external. May generate inefficient code.
Object Module Errors	
Bad Object Module	this file is not recognized as a valid object module created by the assembler. Probably the wrong filename.

3.6.1 Briefmap — Set Map Format

Syntax: **[NO]Briefmap**

Description: This command allows the user to get a full or brief load map. A brief map consists of the "Memory Order Map", which lists memory usage in numeric order, and the "Memory Type Map", which list memory usage by memory type, *e.g.*, BASE, ROM16, *etc.* and the "Total Memory Map", which shows the total RAM, ROM usage. Default is NOBRIEFMAP.

A full map (NOBRIEF), consists of the brief map plus a listing of each section name and where the section is allocated. In addition, the modules that make up the section are listed. Section 3.3 shows these maps.

TABLE 3-2. SUMMARY OF LINKER COMMANDS

COMMAND	FUNCTION	SECTION
[NO]Briefmap	[NO]Brief load map	3.6.1
[NO]Crossref	[NO]Cross reference table in output map file	3.6.2
[NO]Echo	[NO]Echo command file	3.6.3
Format=*type*	Specifies format of output object module	3.6.4
LIBFile=*type*	Library files	3.6.5
LIBDirectory=*directory*	Search other library/directories	3.6.6
[NO]Mapfile[=*file*]	[NO]Map file	3.6.7
[NO]Outputfile[=*file*]	[NO]Absolute object module	3.6.8
Range *type=ranges*	Specifies ranges for section type	3.6.9
Sect *section=addr*	Specifies address or range of section	3.6.10
[NO]Tablesymbols	[NO]List symbol table	3.6.11
[NO]Warnings	[NO]Output warning messages	3.6.12
[NO]Xdirectory	[NO]Search only LIBDirectory directories	3.6.13

3.6.3 Echo — Echo Command Files

Syntax: [NO]Echo

Description: ECHO causes all lines read from a command file to be displayed on the console. A command file is one specified by @*filename*. The default is NOECHO.

3.6.2 Crossref — Cross Reference

Syntax: **[NO]Crossref**

Description: CROSSREF causes a cross reference table to be included in the output map file. This table consists of each symbol and its value along with the names of the module in which it is defined and all the modules in which it is used. Local symbols are shown in the table, if passed by the assembler. The module, in which the symbol is defined, is preceded by a - on the listing. Default is NOCROSSREF.

CROSSREF overrides TABLESYMBOLS. A cross reference listing will not be generated if NOMAPFILE is in effect.

3.6.5 Libfile — Specify Library File To Search

Syntax: **LIBFile =** ***file*****, LF =** ***file***

Description: This command specifies library files that will be used to resolve any undefined externals. If a library module contains a PUBLIC symbol that matches an unresolved external, that module will be automatically loaded by the linker. The library search is performed at the end of the linking process. Multiple libraries will be searched in the order specified.

To search a library at some other point in the load process, the library file must be specified on the invocation line as described in Section 3.2.1.

The default extension for a library file is .lib.

Example: /LF=MATH /LF=FLOAT

3.6.4 Format — Specify Output Format

Syntax: **Format** = *type* [= *strip*]

Description: This command specifies the format of the output object module. *Type* may be "lm" for National Semiconductor load module format, "coff" for a COFF (Common Object File Format) type object module or "hex" for an Intel-hex format. The format type may be specified in either upper or lower case. The default is hex format.

Only a COFF type object module may contain symbols and thus would usually be used for symbolic debugging purposes.

For a format that allows symbolic information, *e.g.*, COFF, the default is to put the symbols in the object module. The optional strip argument may be used to keep symbols out of the object module.

Example:

```
/F=hex
/F=coff=strip
```

3.6.7 Mapfile — Specify Map File

Syntax:	[NO]**Mapfile** [= *file*]
Description:	***Mapfile*** specifies the name of the file to which the load map as well as any symbol or cross reference table will be written. If NOMAPFILE is specified then no map is produced. MAPFILE without a filename will cause the map to be written to the first object filename with an extension of .map. The default is MAPFILE. The default filename extension is .map.
	The BRIEF command is used to specify the kind of map.
Example:	/NOMAP
	/M=test map to file test.map

3.6.8 Outputfile — Specify Output Object File

Syntax: **[NO]Outputfile** [=*file*]

Description: *File* specifies the name of the file that the absolute object module, produced by the linker, will be written. If NOOUTPUTFILE is specified then no file is produced. The default is OUTPUTFILE with default filename.

The format of the output file as well as its default extension is given by the FORMAT command. OUTPUTFILE without a filename will cause the output to be written to the first object filename with an extension depending upon the FORMAT command. A format of lm will use an extension of .lm, a format of COFF will use .cof as an extension and a format of hex will use .hex as an extension.

Example:

/O	default output file
/O=test.abs	output to file test.abs

3.6.9 Range — Specify Memory Ranges

Syntax: **Range memtype=*ranges***

where:

memtype	indicates the memory type and may be BASE, RAM8, RAM16, ROM8, or ROM16.
ranges	is one or more memory ranges. Multiple ranges must be separated by commas. In MS-DOS multiple ranges must be enclosed within parentheses. Each range must be in the form:

```
      low address:high address
   or type
```

Description: The RANGE command allows the user to denote those areas of memory in which the various section memory types will be placed.

If an address range is given, it implies that the "memtype" for the command may reside in this part of memory. *Type* specifies a memory type just like "memtype" and implies that the range may also consist of the ranges for the memory type given by *type*. The default is:

```
ROM16 = 0xF000:0xFFCF
ROM8  = ROM16
BASE  = 0:0xBF
RAM16 = (0x1C0:0x1FF,BASE)
RAM8  = RAM16
```

See Section 3.4 for additional information.

Example:

```
1   /RANGE ROM16=(0x8000:0x83ff , 0xf000:0xffcf)
2   /R ROM8=(0x1000:0x13ff,ROM16)
```

Example 1 tells the linker that ROM16 code can only reside between 0x8000 to 0x83ff and 0xf000 and 0xffcf. In addition, the linker will first attempt to fill the first range (0x8000:0x83ff) before using the second range.

Example 2 says to put any ROM8 code into the range 0x1000 to 0x13ff. If no room exists, then use the ranges specified for memory type ROM16.

3.6.10 Sect — Specify Section Address

Syntax: **Sect name=***addr***, Sect name=***range***, Sect name=***section*

Description: This command is used to specify a starting address or address range for section "name". If *addr* is used, then this section will be placed in memory at the given address. If *range* is used, then the section will be allocated somewhere within that range of addresses. If a previous section name is given, then the address or range used for that section will be used for this one.

Example:

/s one=566	section one starts at 0x566
/s two=0x200:0x300	section two must reside in this range
/S three=two	section three is also between 0x200:0x300

3.6.11 Tablesymbols — Enable Symbol Table

Syntax: [NO]**Tablesymbols**

Description: NOTABLESYMBOLS specifies that no symbol table will be listed in the map file, while TABLESYMBOLS creates a symbol table listing. This command is overridden by the CROSSREF control. The default is NOTABLESYMBOLS.

A symbol table listing will not be generated if NOMAPFILE is in effect.

3.6.12 Warnings — Display Warning Messages

Syntax: **[NO]Warnings**

Description: WARNINGS causes any warning messages to be included in the map file. NOWARNINGS inhibits the messages. The default is WARNINGS.

3.6.13 Xdirectory — Exclude Standard Directories

Syntax: **[NO]Xdirectory**

Description: Normally when searching for a library file, the current, default, and any directories specified via the LIBDIRECTORY command are searched. XDIRECTORY causes only those directories specified by the LIBDIRECTORY command to be searched. This allows files in another directory that have the same name as those in the current directory to be accessed. The default is NOXDIRECTORY.

4.2.5 Help File Search Order

When searching for a helpfile, directories are searched in the following order:

1. current directory
2. default directories
    a. directory specified by environment variable LIBHPC, if it exists
    b. directory specified by environment variable HPC, if it exists
    c. directory \HPC

4.3 LIBRARY ERRORS

Except for those messages listed under Command Warnings in Table 4-1, all the rest of the error conditions will cause the librarian to stop operation with no changes to the library file being operated on.

```
LIBHPC FLOAT SINE,COSINE /REPLACE
LIBHPC TEST  DATASET.1 /DELETE
```

The first command line replaces the object modules sine.obj and cosine.obj in the library *float.lib*.

The second command line deletes the module *dataset.1* from the library *test.lib*.

4.2.2 Object Files and Module Names

An object file is a file generated by *ASMHPC*. This file usually has an extension of .obj. The Module Name by which the librarian stores these files in a library, is the same as the filename without the extension. Thus, an object file of *tangent.obj* would have a module name of tangent within the library.

4.2.3 Library Options

An option is a library command specified on the invocation line in a manner consistent with the operating system. See Appendix B for the library options on a UNIX system. See Appendix C for the library options on a VMS system.

The invocation line options for MS-DOS systems start with a / which may be preceded and followed by white space. Library options are not case sensitive and may be abbreviated to the minimum number of characters as specified in the command descriptions. For example:

```
/ADD  / Backup
```

See Section 4.4 for all the library commands.

4.2.4 Default File Names and Extensions

For those options that require a filename, the name may include a directory path. It may also consist of just a directory path. In this case, the default filename will be used but written to the directory. The default filename is the name of the first object file specified, with any extension removed.

Default extensions depend on the operating system and how the file is specified on the invocation line. For MS-DOS systems, a default extension is always placed on a file unless one is explicitly specified.

If an output filename consists of just a directory, it should always be terminated by a "\" in MS-DOS systems. If not it will be treated as a filename. Thus, /L=txt\ would output the list to file *txt*(assuming *cat.lib* is the input file). /L=txt would output the listing to *txt.lis*.

4.4 LIBRARY COMMANDS

This section describes the library commands. The descriptions show the command name, arguments, and default state. Table 4-2 is a summary of the library commands.

Only one of the primary commands, ADD, DELETE, LIST, REPLACE, or UPDATE may appear on the invocation line at a time.

TABLE 4-2. SUMMARY OF LIBRARY COMMANDS

COMMAND	FUNCTION	SECTION
Add	Place specified object file(s) in library	4.4.1
[NO]Backup	[NO]Backup library file	4.4.2
Delete	Remove module from library	4.4.3
[NO]Echo	[NO]Echo command file	4.4.4
[NO]List[=*file*]	[NO]List all modules in library	4.4.5
Replace	Replace specified object file(s) in library	4.4.6
Update	Replace object file(s) with object file with a later date	4.4.7
[NO]Warnings	[NO]Output warning messages	4.4.8

TABLE 4-1. LIBRARY ERRORS

ERROR	DESCRIPTION
Command Errors	
Invalid Command	The command specified is not a valid librarian command.
Ambiguous Command	This command abbreviation can be more than one command.
Invalid File Name	The file specified is not a valid filename.
File Not Found	The object file cannot be found. Maybe a wrong extension is assumed or it's in a different directory.
Duplicate Module Name	An attempt to ADD a module to a library which already contains a module with this name.
File is not an Object Module	The specified file is not an object module.
File not a Library	The specified file does not look like a library.
Already specified library operation	Only one library operation may be specified for each execution.
No Library Specified	No library was specified on the invocation line.
No Library Operation specified	self explanatory
No Modules to Operate on	An operation other than LIST was specified but no modules were given to operate on.
Duplicate PUBLIC symbol	The specified symbol duplicates a PUBLIC symbol in another module.
Cannot create library file	Library file or backup file can't be created. Check filename.
"NO" not valid for command	Command can't use NO.
Command Warnings	
Module not Found	An attempt to delete a module which is not in the library.
Object Errors	
Bad Object Record	Something is wrong with the object module file.
Checksum Error	An object record has a checksum error. Reassemble the file.
SYNC Error	File or system changed pass 1 to pass 2. Try to load program again.

4.4.2 Backup — Create Backup Library

Syntax: **[NO]Backup**

Description: The librarian always creates a new library so as to avoid any possible damage to the old library. The old library is then renamed with a default extension of .bak. NOBACKUP may be used to request that no backup library be created. In any case, any error causes the library to remain unchanged. The default is BACKUP.

4.4.1 Add — Add Object Module

Syntax: **Add**

Description: This command tells the librarian to place the specified object file(s) into the library. If the module already exists in the library, an error is displayed. An object module has the default extension .obj.

Example:

```
LIBHPC lib1   mod1,mod2,mod3 /add
```

This will add files *mod1.obj*, *mod2.obj* and *mod3.obj* to the library *lib1.lib*. The library will be created if it doesn't already exist. The modules will have module names of mod1, mod2 and mod3.

4.4.4 Echo — Echo Command Files

Syntax: **[NO]Echo**

Description: ECHO causes all lines read from a command file to be displayed on the console. A command file is one specified by @*filename*. The default is NOECHO.

4.4.3 Delete — Delete Object Module

Syntax: **Delete**

Description: DELETE causes the object modules specified to be removed from the library. A warning is given if the module does not exist in the library.

Remember that the module name is the same as the filename without an extension.

Example:

```
LIBHPC math add,sub /delete
```

4.4.6 Replace — Replace Object Module

Syntax:	**Replace**
Description:	This command tells the librarian to replace the specified object file(s) in the library. If the module already exists in the library, it is replaced. If not, it is added to the library. This command is similar to ADD but it is not an error if the module exists in the library.
Example:	LIBHPC lib1 mod1,mod2,mod3 /replace

4.4.5 List — List Library

Syntax: [NO]List [= *file*]

Description: LIST displays a list of all modules in the library. This list will be written to the specified file. If no file is given it will go to the console. The default is NOLIST and the default extension is .lis.

4.4.8 Warnings — Display Warning Messages

Syntax: **[NO]Warnings**

Description: WARNINGS causes any warning messages to be included in the output listing. NOWARNINGS inhibits the messages. The default is WARNINGS.

4.4.7 Update — Replace Object Module If Newer

Syntax:	**Update**
Description:	This command tells the librarian to replace the specified object file(s) in the library, if the new object file has a later date then the one in the library. A module that does not already exist in the library is always added to it.
Example:	`LIBHPC lib1 mod1,mod2,mod3 /update`

Appendix A

ASCII CHARACTER SET IN HEXADECIMAL REPRESENTATION

CHAR.	7-BIT HEX NUMBER	CHAR.	7-BIT HEX NUMBER	CHAR.	7-BIT HEX NUMBER	CHAR.	7-BIT HEX NUMBER
NUL	00	SP	20	@	40	‘	60
SOH	01	!	21	A	41	a	61
STX	02	"	22	B	42	b	62
ETX	03	#	23	C	43	c	63
EOT	04	$	24	D	44	d	64
ENQ	05	%	25	E	45	e	65
ACK	06	&	26	F	46	f	66
BEL	07	'	27	G	47	g	67
BS	08	(	28	H	48	h	68
HT	09	)	29	I	49	i	69
LF	0A	*	2A	J	4A	j	6A
VT	0B	+	2B	K	4B	k	6B
FF	0C	,	2C	L	4C	l	6C
CR	0D	-	2D	M	4D	m	6D
SO	0E	.	2E	N	4E	n	6E
SI	0F	/	2F	O	4F	o	6F
DLE	10	0	30	P	50	p	70
DC1	11	1	31	Q	51	q	71
DC2	12	2	32	R	52	r	72
DC3	13	3	33	S	53	s	73
DC4	14	4	34	T	54	t	74
NAK	15	5	35	U	55	u	75
SYN	16	6	36	V	56	v	76
ETB	17	7	37	W	57	w	77
CAN	18	8	38	X	58	x	78
EM	19	9	39	Y	59	y	79
SUB	1A	:	3A	Z	5A	z	7A
ESC	1B	;	3B	[	5B	{	7B
FS	1C	<	3C	\	5C	|	7C
GS	1D	=	3D	]	5D	}	7D
RS	1E	>	3E	↑	5E	~	7E
US	1F	?	3F	←	5F	DEL, rubout	7F

Appendix B

UNIX INVOCATION AND OPERATIONS

B.1 INTRODUCTION

This appendix describes the invocation and operations for *asmhpc*, *lnhpc* and *libhpc* on a UNIX operating system. See the release letter for installation instructions.

B.2 CROSS-ASSEMBLER INVOCATION AND OPERATION

This section describes the operation of the HPC assembler, *asmhpc*, under a UNIX operating system.

B.2.1 Assembler Invocation

The assembler, *asmhpc*, invocation for a UNIX operating system is as follows:

asmhpc (gives help)

asmhpc [*options*] [*asmfile* [*asmfile* [...]]]

where:	*asmfile*	is the name of a program to assemble. Multiple files may be assembled by specifying multiple files with a space after each filename. Multiple source files is designed to allow the user to include standard definition files without having to specify them in each program.
	options	is a list of assembler options. No options need be specified in which case the assembler defaults will be used. Section B.2.2 "Assembler Options" describes the option syntax.

NOTE: Command files such as @*cmdfile* under MS-DOS can be implemented by `cat cmdfile` under UNIX.

Any errors detected on the invocation line causes the assembler to stop execution and display the part in error with an appropriate message.

Some sample invocation lines for UNIX systems are:

```
asmhpc -L /dev/tty -OT test.asm
asmhpc -cl jack.src
asmhpc -D count=6 -p w=120 /demo/sample.asm
asmhpc -L /demo /demo/sample.asm
asmhpc math.def math.asm
```

B.2.4 Include File Search Order

When searching for a file specified on the .INCLD directive, directories are searched in the following order:

1. current directory
2. directories specified by the -I option
3. default directories
   a. directory specified by environment variable ASMHPC, if it exists
   b. directory specified by environment variable HPC, if it exists
   c. directory /hpc

If the -x option is specified, only directories in category 2 above will be used. Any filename which contains an explicit directory will only be checked for in that directory. No other directories will be searched.

B.2.5 Help File Search Order

When searching for the help file *asmhpc.hlp*, directories are searched in the following order:

1. current directory
2. default directories (as noted in Section B.2.4)

B.2.6 Option Table

Table B-1 shows the mapping between UNIX and MS-DOS invocations.

The -z option in a UNIX system may be used to specify any control. This provides a mechanism to use all the assembler controls without having meaningless single letter options as is the case with many UNIX programs. Thus

```
-c  is equivalent to -z crossref or -z c
```

The argument of the -z option may contain multiple controls separated by commas. For example:

```
-cf is equivalent to -z c,f
```

Although the UNIX options are case sensitive, the argument(s) of the -z option are not. They follow the standard assembler control rules. For consistency, MS-DOS or VMS systems may also use the Z option in a similar fashion. For example:

```
/Z=(c,f)
```

The first command line assembles the file *test.asm* and outputs a listing to the console. No object module or symbol table is produced.

The second command line assembles the file *jack.src* in the current directory and outputs a listing to the default file *jack.lis*. In addition, the object module is written to the file *jack.obj*.

The next example assembles the file *sample.asm* which resides in the directory *demo*. It produces *sample.obj*, which will be in the current directory. The symbol COUNT is defined and given the value 6. Also, the page width is set to 120 characters.

The fourth example is just like the previous one except that the output listing will be written to the same directory as the source file. In this case, the filename is the default which is *sample.lis*.

The last example shows two files being assembled as one assembly file. The program first assembles *math.def*. When the end of file is reached, it then continues the assembly with *math.asm*.

B.2.2 Assembler Options

An invocation line option is an assembler control that is specified in a manner consistent with the operating system. Refer to Section 2.10 for a detailed description of the controls.

Assembler options for a UNIX operating system follow the System V Interface Definition. Each option is preceded by a -. Options are case sensitive. There are no optional arguments, the option either requires an argument, or it doesn't. Multiple options without arguments may be appended together or listed separately. For example:

-sm is equivalent to -s -m

See Section B.2.6 for mapping between UNIX and MS-DOS assembler command line options.

B.2.3 Default File Names and Extensions

For those options that require a filename, the name may include a directory path. It may also consist of just a directory path. In this case the default filename will be used but written to the directory. The default filename is the name of the last source file specified, with any extension removed.

For UNIX operating systems, a default extension is only used when no filename is given.

If an output filename consists of just a directory, it should always be terminated by a "/" in a UNIX operating system. If not, it will be treated as a filename.

B.3 CROSS-LINKER INVOCATION AND OPERATION

This section describes the operation of the HPC linker, *lnhpc*, under a UNIX operating system.

B.3.1 Linker Invocation

The linker, *lnhpc*, invocation for a UNIX system is as follows:

lnhpc (gives help)

lnhpc [*options*] [*objfile*[*objfile*[...]]]

where:	*objfile*	is the name of an object module or library file to load. Multiple files may be linked by specifying multiple files, with a space after each file. The linker determines the file type from the file contents. Only those modules in the library that satisfy undefined externals will be linked. Thus, the order of the libraries when used in this manner is important. In order to use a library, the complete filename is given. No default extension is assumed. For another way of specifying libraries see Section 3.6.5.
	options	is a list of linker commands described in Section 3.2.2. No options need be specified in which case the linker defaults will be used. NOTE: For a UNIX system, the user can use the form `cat cmdfile` on the command line to simulate command files under MS-DOS of the form *@cmdfile*.

Any errors detected on the invocation line causes the linker to stop execution and display the part in error with an appropriate message.

Some sample invocation lines for a UNIX system are:

```
lnhpc  -O /dev/null -t test.obj
lnhpc  -cbm mod1.obj mod2.obj
lnhpc  -s data=0x400 /demo/sample.obj
```

The first command line links the file *test.obj* and outputs the load map to the console. No output object file is produced but a symbol table is listed.

The second command line links the object modules *mod1.obj* and *mod2.obj* and places the output object file into *mod1.hex*. A cross reference table is produced and the brief form of the map is used. The map is written to file *mod1.map*.

The next example links the file *sample.obj*, which resides in directory *demo*. It produces *sample.hex* which will be in the current directory. A section, called data, in the object module is assigned a starting address of 0x400.

TABLE B-1. MAPPING BETWEEN MS-DOS AND UNIX INVOCATION

MS-DOS	UNIX	MS-DOS	UNIX
/Aserrorfile	-a	/MCOmments	-z MCOmments
/Aserrorfile=file	-A file	/NOMCOmments	-z NOMCOmments
/NOAserrorfile	-A /dev/null	/MDefinitions	-z MDefinitions
/CNDDirectives	-z CNDDirectives	/NOMDefinitions	-z NOMDefinitions
/NOCNDDirectives	-z NOCNDDirectives	/MExpansions	-z MExpansions
/CNDLines	-z CNDLines	/NOMExpansions	-z NOMExpansions
/NOCNDLines	-z NOCNDLines	/MLocals	-z MLocals
/COMMentlines	-z COMMentlines	/NOMLocals	-z NOMLocals
/NOCOMMentlines	-z NOCOMMentlines	/MObject	-z MObject
/COMPlexrel	-z COMPlexrel	/NOMObject	-z NOMObject
/NOCOMPlexrel	-z NOCOMPlexrel	/Numberlines	-n
/CONstants	-z CONstants	/NONumberlines	-N
/NOCONstants	-z NOCONstants	/Objectfile	-o
/Crossref	-c	/Objectfile=file	-O file
/NOCrossref	-C	/NOObjectfile	-O /dev/null
/DAtadirectives	-z DAtadirectives	/PAss=number	-P number
/NODAtadirectives	-z NODAtadirectives	/PLength=number	-p l=number
/Define symbol	-D symbol	/PWidth=number	-p w=number
/Define symbol=number	-D symbol=number	/Quick	-q
/ECho		/Remove	-r
/NOECho		/NORemove	-R
/Errorfile	-e	/REStore	-z REStore
/Errorfile=file	-E file	/SAve	-z SAve
/NOErrorfile	-E /dev/null	/SIGnedcompare	-z SIGnedcompare
/Formfeed	-f	/NOSIGnedcompare	-z NOSIGnedcompare
/NOFormfeed	-F	/SIZesymbol=number	-z SIZesymbol=number
/Headings	-h	/Tablesymbols	-t
/NOHeadings	-H	/NOTablesymbols	-T
/ILines	-z ILines	/TABS	-z TABS
/NOILines	-z NOILines	/NOTABS	-z NOTABS
/Include=directory	-I directory	/Undefine symbol	-U symbol
/Listfile	-l	/UPpercase	-z UPpercase
/Listfile=file	-L file	/NOUPpercase	-z NOUPpercase
/NOListfile	-L /dev/null	/Verify	-v
/LOcalsymbols	-z localsymbols	/NOVerify	-V
/NOLOcalsymbols	-z nolocalsymbols	/Warnings	-w
/Masterlist	-m	/NOWarnings	-W
/NOMasterlist	-M	/Xdirectory	-x
/MCAlls	-z MCAlls	/NOXdirectory	-X
/NOMCAlls	-z NOMCalls		

B.3.3 Default File Names and Extension

For those options that require a filename, the name may include a directory path. It may also consist of just a directory path. In this case, the default filename will be used but written to the directory. The default filename is the name of the first object file specified, with any extension removed.

For UNIX systems, a default extension is only used when no filename is given.

If an output filename consists of just a directory, it should always be terminated by a "/" in UNIX systems. If not, it will be treated as a filename.

B.3.4 Library File Search Order

When searching for a file specified by the -l option, directories are searched in the following order:

1. current directory
2. directories specified by the -L option
3. default directories

    a. directory specified by environment variable LNHPC, if it exists
    b. directory specified by environment variable HPC, if it exists
    c. directory /hpc

If the -x option is specified, only directories in category 2 above will be used. Any filename which contains an explicit directory will only be checked for in that directory. No other directories will be searched.

B.3.5 Help File Search Order

When searching for the help file *lnhpc.hlp*, directories are searched in the following order:

1. current directory
2. default directories (as noted in Section B.3.4).

B.4 CROSS-LIBRARIAN INVOCATION AND OPERATION

This section describes the operation of the HPC librarian, *libhpc* under a UNIX operating system.

B.3.2 Linker Options

An option is a linker command specified on the invocation line in a manner consistent with the UNIX operating system.

Linker options for UNIX systems follow the System V Interface Definition. Each option is preceded by a "-". Options are case sensitive. There are no optional arguments, the option either requires an argument, or it doesn't. Multiple options may be appended together or listed separately. For example:

-sm is equivalent to -s -m

See Table B-2 for the mapping of MS-DOS and UNIX linker command line options.

TABLE B-2. MAPPING BETWEEN MS-DOS AND UNIX LINKER COMMANDS

MS-DOS	UNIX	MS-DOS	UNIX
/Briefmap	-b	/NOOutputfile	-O /dev/null
/NOBriefmap	-B	/Range BASE=range	-r BASE=ranges
/Crossref	-c	/Range ROM16=ranges	-r ROM16=ranges
/NOCrossref	-C	/Range ROM8=ranges	-r ROM8=ranges
/ECho		/Range RAM16=ranges	-r RAM16=ranges
/NOECho		/Range RAM8=ranges	-r RAM8=ranges
/Format=type	-f type	/Sect name=addr	-s name=addr
/LIBDirectory=directory	-L directory	/Sect name=range	-s name=range
/LD=directory		/Sect name=section	-s name=section
/LIBFile=libfile	-l libfile	/Tablesymbols	-t
/LF=libfile		/NOTablesymbols	-T
/Mapfile	-m	/Warnings	-w
/Mapfile=file	-M file	/NOWarnings	-W
/NOMapfile	-M /dev/null	/Xdirectory	-x
/Outputfile	-o	/NOXdirectory	-X
/Outputfile=file	-O file		

See Table B-3 for the mapping between MS-DOS and UNIX library command line options.

TABLE B-3. MAPPING BETWEEN MS-DOS AND UNIX LIBRARY COMMANDS

MS-DOS	UNIX	MS-DOS	UNIX
/Add	-a	/List=file	-L file
/Backup	-b	/NOList	-L /dev/null
/NOBackup	-B	/Replace	-r
/Delete	-d	/Update	-u
/ECho		/Warnings	-w
/NOECho		/NOWarnings	-W
/List	-l		

B.4.4 Default File Names and Extensions

For those options that require a filename, the name may include a directory path. It may also consist of just a directory path. In this case, the default filename will be used but written to the directory. The default filename is the name of the first object file specified, with any extension removed.

For UNIX systems, a default extension is only used when no filename is given.

If an output filename consists of just a directory, it should always be terminated by a "/" in UNIX systems. If not it will be treated as a filename.

B.4.5 Help File Search Order

When searching for a help file, directories are searched in the following order:

1. current directory
2. default directories
    a. directory specified by environment variable LIBHPC, if it exists.
    b. directory specified by environment variable HPC, if it exists.
    c. directory /hpc

B.4.1 Librarian Invocation

The librarian, *libhpc*, invocation for UNIX systems is as follows:

libhpc (gives help)

libhpc *options libfile* [*name* [*,name...*]]

where:	*libfile*	is the name of a library to process. If it does not already exist, then a new library will be created.
	options	is a list of library commands described in Section B.4.3.
	name	is a list of one or more object files that that will be processed by *libhpc*.

NOTE: For UNIX systems, the user can use the form `cat cmdfile` on the command line to simulate *@cmdfile* under MS-DOS.

Any errors detected on the invocation line causes the librarian to stop execution and display the part in error with an appropriate message.

For UNIX systems sample command lines are:

```
libhpc -r float.lib  sine.obj ,cosine.obj
libhpc -d test.lib   dataset.1
```

The first command line replaces the object modules *sine.obj* and *cosine.obj* in the library *float.lib*.

The second command line deletes the module *dataset.1* from the library *test.lib*.

B.4.2 Object Files and Module Names

An object file is one which was generated by *asmhpc*. This file usually has an extension of .obj. The Module Name by which the librarian stores these files in a library is the same as the filename without the extension. Thus an object file of *tangent.obj* would have a module name of tangent within the library. On UNIX, module names are case sensitive.

B.4.3 Library Options

An option is a library command specified on the invocation line in a manner consistent with the UNIX operating system. Library options for UNIX systems follow the System V Interface Definition. Each option is preceded by a "-". Options are case sensitive. There are no optional arguments, the option either requires an argument, or it doesn't. Multiple options may be appended together or listed separately. For example:

-ub is equivalent to -u -b

Appendix C

VMS INVOCATION AND OPERATIONS

C.1 INTRODUCTION

This appendix describes the invocation and operations for *ASMHPC*, *LNHPC* and *LIBHPC* on a VMS operating system. See the release letter for installation instructions.

C.2 CROSS-ASSEMBLER INVOCATION AND OPERATION

This section describes the operation of the HPC assembler, *ASMHPC*, under a VMS operating system.

C.2.1 Assembler Invocation

The assembler, *ASMHPC*, invocation for VMS operating systems is as follows:

ASMHPC (gives help)

ASMHPC [*asmfile* [,*asmfile* [,...]]] [@*cmdfile*] [@*cmdfile...*] [*options*]

where:	*asmfile*	is the name of a program to assemble. Multiple files may be assembled by joining them with a "," for VMS systems. Multiple source files is designed to allow the user to include standard definition files without having to specify them in each program.
	options	is a list of assembler options. No options need be specified in which case the assembler defaults will be used. Section C.2.2 "Assembler Options" describes the option syntax.
	cmdfile	is the name of a file that contains additional invocation line source file names and/or options. This file has a default extension of .COM for VMS systems. For example, if the file *a.com* contains:

```
test /a
/o
```

and file *b.com* contains

```
/l=file
```

then the command

```
ASMHPC @a /e @b
```

C.2.3 Default File Names and Extensions

For those options that require a file name, the name may include a directory path. It may also consist of just a directory path. In this case, the default file name will be used but written to the directory. The default file name is the name of the last source file specified, with any extension removed.

For VMS operating systems, a default extension is always placed on a file unless one is explicitly specified.

C.2.4 Include File Search Order

When searching for a file specified on the .INCLD directive, directories are searched in the following order:

1. current directory
2. directories specified by the /I option
3. default directories
    a. directory specified by logical name ASMHPC, if it exists
    b. directory specified by logical name HPC, if it exists
    c. directory specified by logical name SYS$HPC

If the /X option is specified, only directories in category 2 above will be used. Any file name which contains an explicit directory will only be checked for in that directory. No other directories will be searched.

C.2.5 Help File Search Order

When searching for the help file *ASMHPC.HLP*, directories are searched in the following order:

1. current directory
2. default directories (as noted in Section C.2.4)

C.2.6 Option Table

Table C-1 shows mapping of MS-DOS and VMS invocation. Note that in addition to the shown options, a /Z option is provided, whose arguments are controls exactly as specified in the assembler section with the same abbreviations. Thus, /LISTFILE /CROSSREF is equivalent to /Z=LISTFILE /Z=C or /Z=(L,C). This same /Z option is available on MS-DOS, so invocation options can be the same on both systems. A similar -z option is available on the UNIX system.

is equivalent to

```
ASMHPC test /a /o /e /l=file
```

Any errors detected on the invocation line causes the assembler to stop execution and display the part in error with an appropriate message.

Some sample invocation lines for VMS systems are:

```
ASMHPC TEST /L=TT /NOOBJ/NOTABLE
ASMHPC JACK.SRC /CROSS /L
ASMHPC [DEMO]SAMPLE /D=COUNT=6 /PW=120
ASMHPC [DEMO]SAMPLE /L=[DEMO]
ASMHPC MATH.DEF,MATH
```

The first command line assembles the file *TEST.ASM* and outputs a listing to the console. No object module or symbol table is produced.

The second command line assembles the file *JACK.SRC* in the current directory and outputs a listing to the default file *JACK.LIS*. In addition, the object module is written to the file *JACK.OBJ*.

The next example assembles the file *SAMPLE.ASM* which resides in the directory *DEMO*. It produces *SAMPLE.OBJ*, which will be in the current directory. The symbol COUNT is defined and given the value 6. Also, the page width is set to 120 characters.

The fourth example is just like the previous one except that the output listing will be written to the same directory as the source file. In this case, the file name is the default which is *SAMPLE.LIS*.

The last example shows two files being assembled as one assembly file. The program first assembles *MATH.DEF*. When the end of file is reached, the program continues the assembly with *MATH.ASM*.

C.2.2 Assembler Options

An invocation line option is an assembler control that is specified in a manner consistent with the operating system. See Section 2.10 for a detailed description of the assembler controls.

Assembler options for the VMS operating system start with a / which may be preceded and followed by white space. Options are not case sensitive and may be abbreviated to the minimum number of characters without being ambiguous. Thus, most options will usually be at least two characters in length. See Section C.2.6 for the mapping between VMS and MS-DOS invocation line options.

C.3 CROSS-LINKER INVOCATION AND OPERATION

This section describes the operation of the HPC linker, *LNHPC* under a VMS operating system.

C.3.1 Linker Invocation

The linker, *LNHPC*, invocation for a VMS system is as follows:

LNHPC (gives help)

LNHPC [*objfile* [,*objfile* [,...]]] [@*cmdfile*][@*cmdfile...*] [*options*]

where:

objfile — is the name of an object module or library file to load. Multiple files may be linked by joining them with a "," for VMS systems.

The linker determines the file type from the file contents. Only those modules in the library that satisfy undefined externals will be linked. Thus, the order of the libraries when used in this manner is important. In order to use a library, the complete filename is given. No default extension is assumed. For another way of specifying libraries see Section 3.6.5.

options — is a list of linker commands described in Section 3.2.2. No options need be specified in which case the linker defaults will be used.

cmdfile — is the name of a file that contains additional invocation line object file names and/or options. This file has a default extension of .COM for a VMS system. For example, if the file *a.com* contains:

```
test
/table
```

and file *b.com* contains

```
/o=testdata
```

then the command

LNHPC @a /d @b

is equivalent to

```
LNHPC test /table /d /o=testdata
```

Any errors detected on the invocation line causes the linker to stop execution and display the part in error with an appropriate message.

Some sample invocation lines for a VMS system are:

```
LNHPC  TEST  /NOOUTPUTFILE /TABLE /MAP=TT
LNHPC  MOD1,MOD2  /CROSS /BRIEF /MAP
LNHPC  [DEMO]SAMPLE /SECT=data=0x400
```

TABLE C-1. MAPPING BETWEEN MS-DOS AND VMS INVOCATION

MS-DOS	VMS	MS-DOS	VMS
/Aserrorfile	/ASERRORFILE	/MCOmments	/MCOMMENTS
/Aserrorfile=file	/ASERRORFILE=file	/NOMCOmments	/NOMCOMMENTS
/NOAserrorfile	/NOASERRORFILE	/MDefinitions	/MDEFINITIONS
/CNDDirectives	/CNDDIRECTIVES	/NOMDefinitions	/NOMDEFINITIONS
/NOCNDDirectives	/NOCNDDIRECTIVES	/MExpansions	/MEXPANSIONS
/CNDLines	/CNDLINES	/NOMExpansions	/NOMEXPANSIONS
/NOCNDLines	/NOCNDLINES	/MLocals	/MLOCALS
/COMMentlines	/COMMENTLINES	/NOMLocals	/NOMLOCALS
/NOCOMMentlines	/NOCOMMENTLINES	/MObject	/MOBJECT
/COMPlexrel	/COMPLEXREL	/NOMObject	/NOMOBJECT
/NOCOMPlexrel	/NOCOMPLEXREL	/Numberlines	/NUMBERLINES
/CONstants	/CONSTANTS	/NONumberlines	/NONUMBERLINES
/NOCONstants	/NOCONSTANTS	/Objectfile	/OBJECTFILE
/Crossref	/CROSSREF	/Objectfile=file	/OBJECTFILE=file
/NOCrossref	/NOCROSSREF	/NOObjectfile	/NOOBJECTFILE
/DAtadirectives	/DATADIRECTIVES	/PAss=number	/PASS=number
/NODAtadirectives	/NODATADIRECTIVES	/PLength=number	/PLENGTH=number
/Define symbol	/DEFINE=symbol	/PWidth=number	/PWIDTH=number
/Define symbol=number	/DEFINE=symbol=number	/Quick	/QUICK
/ECho	/ECHO	/REMove	/REMOVE
/NOECho	/NOECHO	/REStore	/RESTORE
/Errorfile	/ERRORFILE	/NOREMove	/NOREMOVE
/Errorfile=file	/ERRORFILE=file	/SAve	/SAVE
/NOErrorfile	/NOERRORFILE	/SIGnedcompare	/SIGNEDCOMPARE
/Formfeed	/FORMFEED	/NOSIGnedcompare	/NOSIGNEDCOMPARE
/NOFormfeed	/NOFORMFEED	/SIZesymbol=number	/SIZESYMBOL=number
/Headings	/HEADINGS	/Tablesymbols	/TABLESYMBOLS
/NOHeadings	/NOHEADINGS	/NOTablesymbols	/NOTABLESYMBOLS
/ILines	/ILINES	/TABS	/TABS
/NOILines	/NOILINES	/NOTABS	/NOTABS
/Include=directory	/INCLUDE=directory	/Undefine symbol	/UNDEFINE=symbol
/Listfile	/LISTFILE	/UPpercase	/UPPERCASE
/Listfile=file	/LISTFILE=file	/NOUPpercase	/NOUPPERCASE
/NOListfile	/NOLISTFILE	/Verify	/VERIFY
/LOcalsymbols	/LOCALSYMBOLS	/NOVerify	/NOVERIFY
/NOLOcalsymbols	/NOLOCALSYMBOLS	/Warnings	/WARNINGS
/Masterlist	/MASTERFILE	/NOWarnings	/NOWARNINGS
/NOMasterlist	/NOMASTERFILE	/Xdirectory	/XDIRECTORY
/MCAlls	/MCALLS	/NOXdirectory	/NOXDIRECTORY
/NOMCAlls	/NOMCALLS		

TABLE C-2. MAPPING BETWEEN MS-DOS AND VMS LINKER COMMANDS

MS-DOS	VMS	MS-DOS	VMS
/Briefmap	/BRIEFMAP	/NOOutputfile	/OUTPUTFILE
/NOBriefmap	/NOBRIEFMAP	/Range BASE=range	/RANGE=BASE=ranges
/Crossref	/CROSSREF	/Range ROM16=ranges	/RANGE=ROM16=ranges
/NOCrossref	/NOCROSSREF	/Range ROM8=ranges	/RANGE=ROM8=ranges
/ECho	/ECHO	/Range RAM16=ranges	/RANGE=RAM16=ranges
/NOECho	/NOECHO	/Range RAM8=ranges	/RANGE=RAM8=ranges
/Format=type	/FORMAT=type	/Sect name=addr	/SECT=name=addr
/LIBDirectory=directory	/LIBDIRECTORY=directory	/Sect name=range	/SECT=name=range
/LD=directory	/LD=directory	/Sect name=section	/SECT=name=section
/LIBFile=libfile	/LIBFILE=libfile	/Tablesymbols	/TABLESYMBOLS
/LF=libfile	/LF=libfile	/NOTablesymbols	/NOTABLESYMBOLS
/Mapfile	/MAPFILE	/Warnings	/WARNINGS
/Mapfile=file	/MAPFILE=file	/NOWarnings	/NOWARNINGS
/NOMapfile	/NOMAPFILE	/Xdirectory	/XDIRECTORY
/Outputfile	/OUTPUTFILE	/NOXdirectory	/NOXDIRECTORY
/Outputfile=file	/OUTPUTFILE=file		

C.3.5 Help File Search Order

When searching for the help file *LNHPC.HLP*, directories are searched in the following order:

1. current directory
2. default directories (as noted in Section C.3.4)

C.4 CROSS-LIBRARIAN INVOCATION AND OPERATION

This section describes the operation of the HPC librarian, *LIBHPC* for a VMS operating system.

The first command line links the file *TEST.OBJ* and outputs the load map to the console. No output object file is produced but a symbol table is listed.

The second command line links the object modules MOD1.OBJ and MOD2.OBJ and places the output object file into *MOD1.HEX*. A cross reference table is produced and the brief form of the map is used. The map is written to the file *MOD1.MAP*.

The next example links the file *SAMPLE.OBJ*, which resides in directory *DEMO*. It produces *SAMPLE.HEX* which will be in the current directory. A section, called data, in the object module is assigned a starting address of 0x400.

C.3.2 Linker Options

An option is a linker command specified on the invocation line in a manner consistent with the VMS operating system.

```
/CROSSREF / MAP
```

Linker options for VMS operating systems start with a / which may be preceded and followed by white space. Options are not case sensitive and may be abbreviated to the minimum number of characters without being ambiguous. See Table C-2 for the mapping of MS-DOS and VMS linker invocation line options.

C.3.3 Default File Names and Extension

For those options that require a file name, the name may include a directory path. It may also consist of just a directory path. In this case, the default file name will be used but written to the directory. The default file name is the name of the first object file specified, with any extension removed.

For VMS systems, a default extension is always placed on a file unless one is explicitly specified.

C.3.4 Library File Search Order

When searching for a file specified by the /LIBFILE option, directories are searched in the following order:

1. current directory
2. directories specified by the /LIBDIRECTORY option
3. default directories
    a. directory specified by logical name LNHPC, if it exists
    b. directory specified by logical name HPC, if it exists
    c. directory specified by logical name SYS$HPC

If the /X option is specified, only directories in category 2 above will be used. Any file name which contains an explicit directory will only be checked for in that directory. No other directories will be searched.

C.4.3 Library Options

An option is a library command specified on the invocation line in a manner consistent with the VMS operating system. Library options for VMS systems start with a / which may be preceded and followed by white space. Options are not case sensitive and may be abbreviated to the minimum number of characters without being ambiguous. See Table C-3 for mapping between MS-DOS and VMS library invocation line options.

TABLE C-3. MAPPING BETWEEN MS-DOS AND VMS LIBRARIAN COMMANDS

MS-DOS	VMS	MS-DOS	VMS
/Add	/ADD	/List=file	/LIST=file
/Backup	/BACKUP	/NOList	/NOLIST
/NOBackup	/NOBACKUP	/Replace	/REPLACE
/Delete	/DELETE	/Update	/UPDATE
/ECho	/ECHO	/Warnings	/WARNINGS
/NOECho	/NOECHO	/NOWarnings	/NOWARNINGS
/List	/LIST		

C.4.4 Default File Names and Extensions

For those options that require a file name, the name may include a directory path. It may also consist of just a directory path. In this case, the default file name will be used but written to the directory. The default file name is the name of the first object file file specified, with any extension removed.

For VMS systems, a default extension is always placed on a file unless one is explicitly specified.

C.4.5 Help File Search Order

When searching for a help file, directories are searched in the following order:

1. current directory
2. default directories
   a. directory specified by logical name LIBHPC, if it exists
   b. directory specified by logical name HPC, if it exists
   c. directory specified by logical name SYS$HPC

C.4.1 Librarian Invocation

The invocation for VMS systems is as follows:

LIBHPC (gives help)

LIBHPC *libfile* [*name* [*,name...*]] [*@cmdfile...*] *options*

where:	*libfile*	is the name of a library to process. If it does not already exist, then a new library will be created.
	options	is a list of library commands described in Section C.4.3.
	name	is a list of one or more object files that that will be processed by *LIBHPC*.
	cmdfile	is the name of a file that contains additional invocation line object file names and/or options. This file has a default extension of .COM for VMS systems.

Any errors detected on the invocation line causes the librarian to stop execution and display the part in error with an appropriate message.

Some sample invocation lines for VMS systems are:

```
LIBHPC  FLOAT  SINE,COSINE  /REPLACE
LIBHPC  TEST   DATASET.1  /DELETE
```

The first command line replaces the object modules *SINE.OBJ* and *COSINE.OBJ* in the library *FLOAT.LIB*.

The second command line deletes the module *DATASET.1* from the library *TEST.LIB*.

C.4.2 Object Files and Module Names

An object file is a file generated by *ASMHPC*. This file usually has an extension of .OBJ. The Module Name by which the librarian stores these files in a library, is the same as the file name without the extension. Thus, an object file of *tangent.obj* would have a module name of tangent within the library.

INDEX

National Semiconductor

MICROCOMPUTER SYSTEMS DIVISION

READER'S COMMENT FORM

In the interest of improving our documentation, National Semiconductor invites your comments on this manual.

Please restrict your comments to the documentation. Technical Support may be contacted at:

(800) 538-1866 - U.S. non CA
(800) 672-1811 - CA only
(800) 223-3248 - Canada only

Please rate this document according to the following categories. Include your comments below.

	EXCELLENT	GOOD	ADEQUATE	FAIR	POOR
Readability (style)	☐	☐	☐	☐	☐
Technical Accuracy	☐	☐	☐	☐	☐
Fulfills Needs	☐	☐	☐	☐	☐
Organization	☐	☐	☐	☐	☐
Presentation (format)	☐	☐	☐	☐	☐
Depth of Coverage	☐	☐	☐	☐	☐
Overall Quality	☐	☐	☐	☐	☐

NAME ______________________ DATE __________

TITLE ______________________

COMPANY NAME/DEPARTMENT ______________________

ADDRESS ______________________

CITY ______________ STATE __________ ZIP ________

Do you require a response? ☐ Yes ☐ No PHONE ______________

Comments:

____

____

____

____

____

____

FOLD, STAPLE, AND MAIL

424410836-001A

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 409 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

National Semiconductor Corporation
Microcomputer Systems Division
Technical Publications Dept. 8278, M/S 7C261
2900 Semiconductor Drive
P.O. Box 58090
Santa Clara, CA 95052-9968